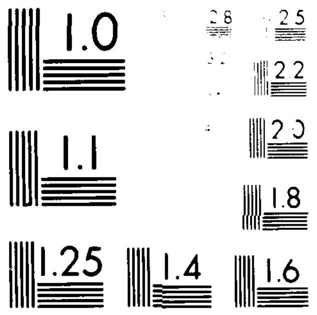
AD-A116 360 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 17/2
SIGMA FINAL REPORT: VOLUME V, PART 4. THE DESIGN OF SIGMA.(U)
MAY 82 R STOTZ, D WILCZYNSKI, S FINKEL DAHC15-72-C-0308
UNCLASSIFIED ISI/RR-82-95-VOL-5-PT-4 NL

1 of 2
AD A
6360

MME

MME



MICROCOPY RESOLUTION TEST CHART

[illegible]

12

MME Final Report
Volume V, Part 4
ISI/RR-82-95

MME

MILITARY MESSAGE EXPERIMENT

SIGMA Final Report: The Design of SIGMA

Robert Stotz
David Wilczynski
Steven Finkel
Robert Lingard
Donald Oestreicher
Leroy Richardson
Ronald Tugender

3

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-82-95	2. GOVT ACCESSION NO. [illegible]	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* SIGMA Final Report: The Design of SIGMA		5. TYPE OF REPORT & PERIOD COVERED Final Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert Stotz, David Wilczynski, Steven Finkel, Robert Lingard, Donald Oestreicher, Leroy Richardson, Ronald Tugender		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 1982
		13. NUMBER OF PAGES 132
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

....

DTIC ELECTE JUN 2 9 1982 S D H

18. SUPPLEMENTARY NOTES

....

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Message Experiment, nonprofessional computer users, SIGMA software architecture, SIGMA message service, TENEX, terminal-based message service, user interface

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This part describes the way the SIGMA system works. The individual components of the system, the main data structures, the communication and data paths, and some unifying concepts are identified and described. SIGMA has many parts, some operating asynchronously, some under user control, some not, some interacting in complex and subtle ways, some very simply. All these factors contribute to the complexity of documenting the design.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

MME Final Report
Volume V, Part 4
ISI/RR-82-95

MME

MILITARY MESSAGE EXPERIMENT

SIGMA Final Report: The Design of SIGMA

Robert Stotz
David Wilczynski
Steven Finkel
Robert Lingard
Donald Oestreicher
Leroy Richardson
Ronald Tugender

This research is supported by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C 0308. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

CONTENTS

PREFACE

This document is Part Four of Volume V of the MME Final Report. The volumes of the MME Final Report and their topics are:

Volume I	Executive Summary
II	Final Report
III	User View
IV	Message System Utility
V	SIGMA Final Report
VI	Data Analysis and Discussion
VII	Training

The opinions expressed in this report are those of the authors and do not necessarily represent those of USC/Information Sciences Institute or the MME project sponsors. Readers interested in obtaining the remaining volumes of the MME Final Report should contact:

Naval Research Laboratory
Washington, DC 20375
Attn: Code 7590

PART FOUR:

THE DESIGN OF SIGMA

4.1 INTRODUCTION--THE INTENT OF THIS PART

This part describes the way the SIGMA system works. As such, the individual components of the system, the main data structures, the communication and data paths, and some unifying concepts are identified and described. The treatments are not uniform; some sections are written in much more detail than others. Many factors led to this: time and space constraints, the availability of other documentation, the author, etc. The reader should not make any value judgments based on the length of any particular section.

The unstructured, nonhierarchical form of this part of the *SIGMA Final Report* reflects the architecture of the SIGMA system itself. SIGMA has many parts, some operating asynchronously, some under user control, some not, some interacting in complex and subtle ways, some very simply. All these factors contribute to the complexity of documenting the design.

We should warn the reader that no single person knows all the "ins and outs" of SIGMA; this document will not produce such a person. It is intended to be a guide to the system from which the reader must glean what he needs. The software architecture section gives an overview of the whole system; for the casual reader that may be enough. To the more serious reader we suggest reading this whole part through once before studying it in any detail.

4.2 SIGMA SOFTWARE ARCHITECTURE

Figure 4-1 shows the overall organization of the SIGMA service. Each on-line SIGMA user is serviced by a *user job*, an instance of which is created for each user accessing the service. A user enters his instructions and data and receives SIGMA's responses through the *MME Terminal*[39], a specially modified Hewlett-Packard CRT terminal with microcode designed for (though independent of) the SIGMA application. The user job accepts, interprets, and executes the user's instructions and manages the display of results on the user's terminal. As seen in Figure 4-2, the user job is composed of five major components. Each component consists of a separate TENEX fork (see section 4.3) except the Terminal Driver, which has two forks. The Command Language Processor (CLP) reads command input, parses it, builds a command specification called an Execution Request Block (ERB), and passes it to the Functional Module (FM) through a protocol called EC99. The FM is responsible for the actual execution of commands, and thus it has two main tasks: to manage the display of information in the terminal, and to manipulate SIGMA's objects. The former task is performed by a module called the Virtual Terminal (VT), which builds and maintains display lists for the terminal. The latter function is done by the FM directly for text objects and selectors, and by two special purpose modules called the Folder Module (FACMOD) and Message Module (MSGMOD) for folders and messages, respectively. Because of address space limitations, FACMOD and MSGMOD are located in separate TENEX forks and require a special Inter-Fork Communication Protocol (IFCP) to communicate with the FM. Communication to and from the terminal is handled by the Terminal Driver, which for control purposes consists of a Receiver fork and a Driver fork. The terminal/user job activities constitute SIGMA's *foreground* processing component.

Activities involving management of the shared databases, off-line processing, and noninteractive functions constitute SIGMA's *background* component. These operations are performed by seven *daemons*, each an independent processor designed to handle a logical subset of the background tasks. Their responsibilities include database management (Message, Folder, Citation daemons), handling of external media including the AUTODIN network through the LDMX interface computer (Reception, Message, Hardcopy, Archive daemons), and various utility functions (Maintenance daemon). The daemons are free-running programs that need no interaction with a human in normal circumstances.

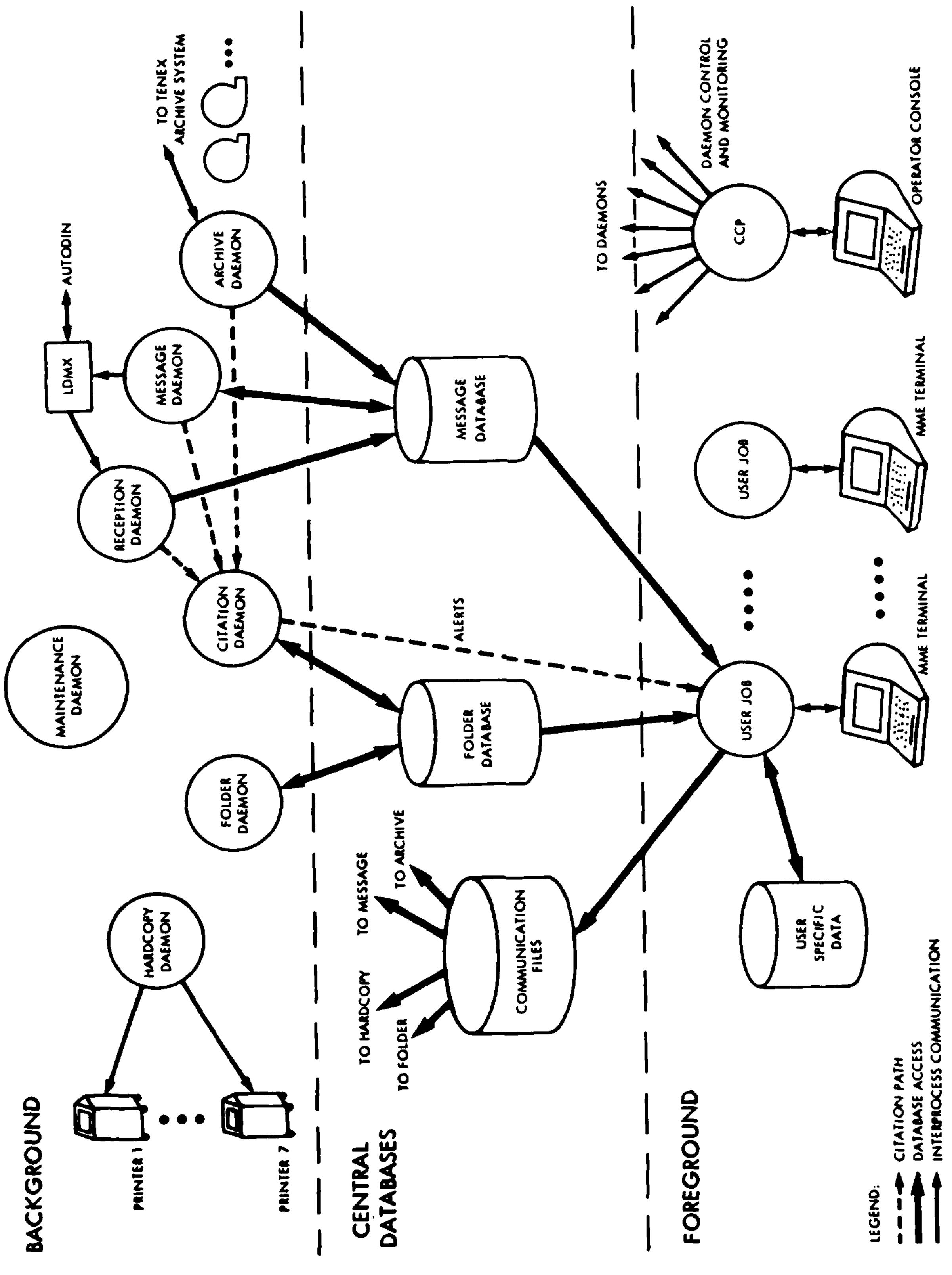


Figure 4-1: SIGMA architecture

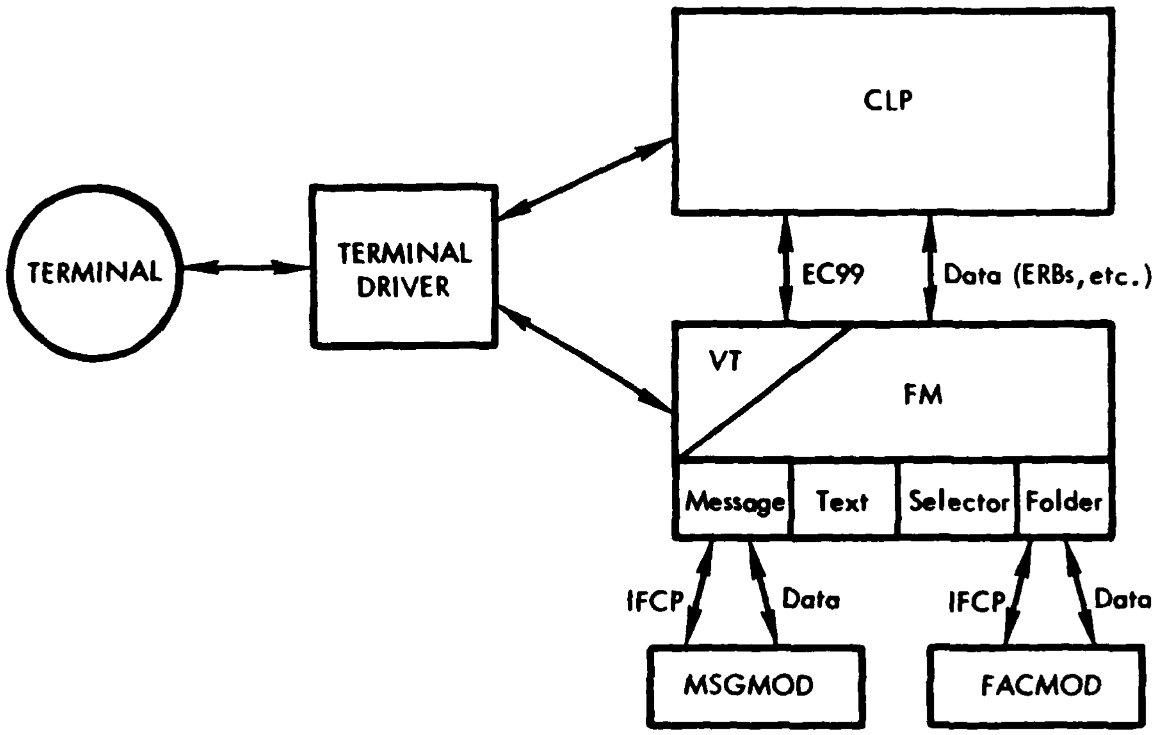


Figure 4-2: User job configuration

Each of the daemons has a structure as depicted in Figure 4-3. A *Processor Controller* (PC), which is exactly the same for each daemon, takes care of initialization, error detection, and reporting. The actual execution of daemon functions is performed by a *Processor* (P) component. This is a separate TENEX fork and is different for each daemon. Each Processor, however, does contain a module which is logically a part of the PC (PC-P interface). For daemons which need to access the SIGMA databases (i.e., the Message, Folder, Reception, and Citation daemons) an access module will also be included.

The daemons are controlled and monitored by an operator through the *Configuration Control Program* (CCP). The CCP communicates with the PC for each daemon through a file (Global State file). The PC reads this file and passes any outside requests (e.g., a request to halt) to the Processor. The Processor then attempts to satisfy the requests at the earliest convenient time. Errors which occur in the daemon are reported back to the CCP via an Error Queue. Those daemons which are driven by requests from the User Jobs receive their instructions through a Request Queue. See section 4.13.5, page 4-102, for a discussion of the Queue Package.

SIGMA's database organization supports its two primary object types, *messages* and *folders*. SIGMA messages contain the text of message traffic, as well as control, status, and annotation information necessary for their handling. Each message is stored once in the database, regardless of the number of recipients. Users receive abstracts of messages (including pointers to the messages), called *citations*, which are stored in folders.

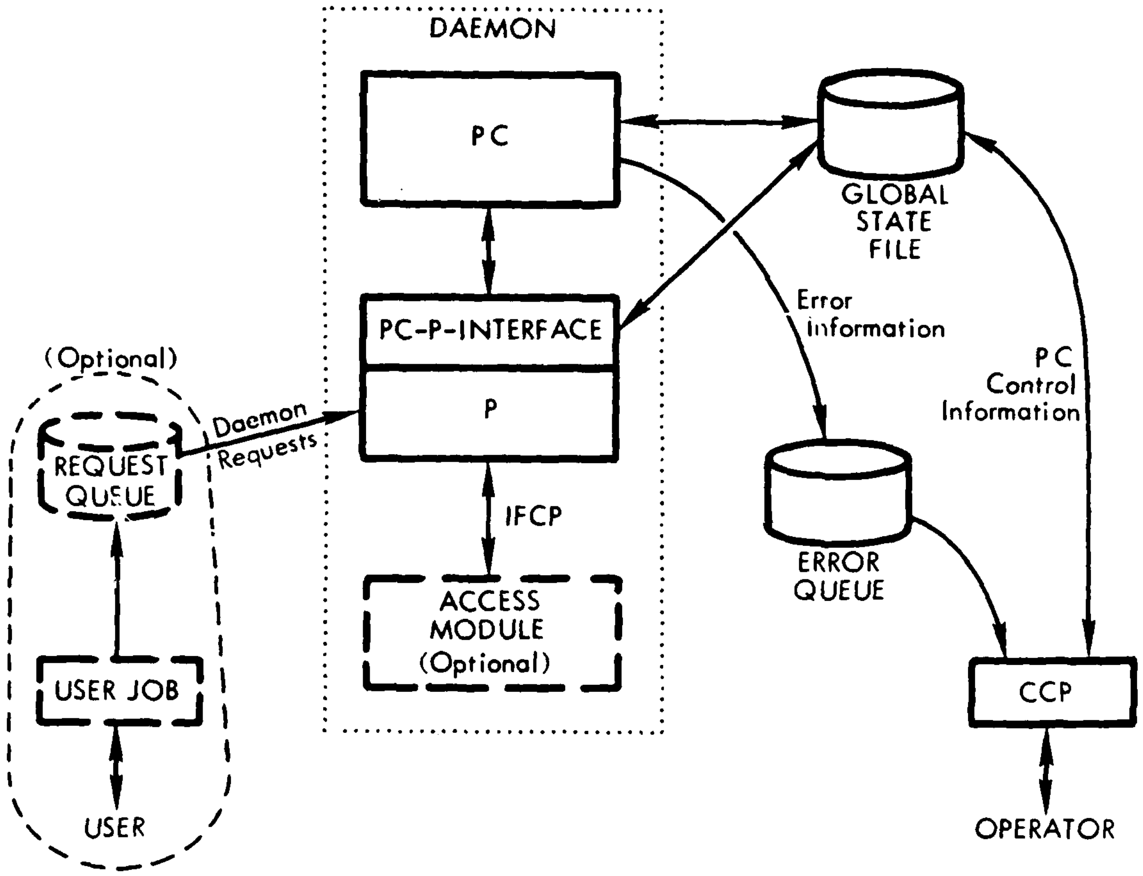


Figure 4-3: Daemon configuration

Folders thus supply an index to the messages and provide the facilities for organizing and retrieving messages and mechanisms for selecting and processing classes of messages. These folders may be shared by many users.

4.2.1 Motivating Factors

The motivation for SIGMA's architecture came from the need to satisfy message handling requirements at CINCPAC [17], which we found to be quite different from those of our own research environment. The following list summarizes the main factors that drove our design.

Message volume

At CINCPAC the incoming message traffic numbers over 1000 per day. The nature of military messages is such that an average of 40 paper copies of each message is made and distributed throughout the headquarters. The CINCPAC community estimated that an automated system should keep the messages on-line for 30 days, and available from archive for a year.

Data sharing

Files of messages are kept for both public and private use. Messages deserving wide distribution are put onto daily readboards, which are then passed around among offices. Often someone will make Xerox copies of messages from a readboard for his personal files. He may highlight or annotate the messages for himself or others.

Coordination

AUTODIN messages are considered formal statements of the commanding officer and sent only in his name. Typically a junior officer will draft a message and pass it through a list of reviewers for coordination before it is released by an authorized senior officer. Each reviewer makes his changes and comments before returning it to the drafter. The drafter then edits the message appropriately and sends it out for further review, if necessary, or on for release.

Single application

The automated system for the MME would deal only with the message handling domain; its users would interact with no other systems on the computer. This requirement forced us to carefully define the complete range of tasks implied by message processing.

Computer-naive users

Even though the users of the automated system would be proficient in their message handling duties, they would not be familiar with computer systems. In addition, their schedules would make extensive classroom training impossible. The system would have to be easy to use and as self-instructive as possible.

Expandable

To allow expansion and redundant access we originally planned to make the message system distributed. However, as the MME focused on the CINCPAC environment, the distributed notions were considered secondary and never implemented. Still, some of the architectural decisions made for SIGMA were influenced by this goal.

Response characteristics

In a distributed environment the MME computer could be very far from the message users. The system had to be configured to provide consistent, acceptable response regardless of system load or distance between users and host.

4.2.2 Architectural Response

The primary purpose of the MME was to determine the effectiveness of interactive message processing in a military environment. Our system would have to be flexible enough to adapt to a changing understanding of the problem and additional requirements imposed by the user community. Sizing, performance, and cost were important, to be sure, but the system did not need to be cost-justified itself, merely sufficient to gain understanding of all these issues. The following architectural features reflect our efforts to satisfy the motivating factors in a flexible, evolutionary system.

Single copy of messages

SIGMA maintains each message as a single shared data file. This organization supports the coordination process as well as reducing the space problems which could exist because of the high fan-out. A message is physically stored as a set of fields; its viewer-dependent display is dynamically constructed by the user job.

Citations

A citation is an abstract of a message. Instead of messages, SIGMA replicates citations, many times smaller than messages, as the communication vehicle. A citation contains enough information--the sender, subject, date/time stamp, etc.--so that the message behind it is identifiable. It also specifies the nature of transmission, whether a message is incoming, for coordination, retrieved from archive, etc. The citation concept forms the basis for folders, archive, and Alert processing.

Folders

A folder is a collection of citations, that is, a file of summaries of messages related in some way. In addition to serving as personal file storage, folders also serve as readboards, action logs, and Datefiles (shared system folders organized by date). The folder serves the user not only as an organizational structure for messages but as a "browsing" media for quick perusal of message activity. Folders form the basis for message database searching and retrieval. The data present in folder entries (citations) can be used as selection criteria in finding classes of messages.

Central data management

Messages and folders are stored in central databases maintained by daemons. Messages that are created in the user job are put in the database by the Message daemon. Some of the Message daemon's other functions include transmitting released messages (back through the LDMX for AUTODIN messages), producing citations for the Citation daemon to deliver (on message creation, forwarding, transmission, etc.), and handling the coordination process. The Folder daemon processes user requests to create, update, or delete folders.

The Citation daemon puts citations into a recipient's PENDING file, a special folder assigned to each user. An on-line user is made immediately aware of new citations through an Alert facility, a communication channel which connects the Citation daemon to the user job.

Since folders and messages can be read and modified simultaneously by several users, a method for maintaining order and consistency had to be incorporated. The daemons provide the mechanism for sequential assimilation of user modifications to folders and messages [43].

Foreground-background

In addition to providing the means for database management, the user job/daemon split addresses the problems of distributing computing cycles in an environment sensitive to response time. The response time burden caused by user jobs competing for time-shared cycles was eased by queuing noninteractive work for sequential processing by the daemons, whose load on the system could be independently controlled by the system scheduler.

Archival

The message volume and on-line storage limitations necessitate a fluid archival scheme. The separation of citations and messages means that users still have pointers to messages (via folders entries) even if the messages have been archived on magnetic tape. With the folders on-line, the user still has all the search mechanisms available to him whether the associated messages are archived or not. When a user asks to see a message that is archived, the user job issues a retrieval request to the Archive daemon. The daemon makes a retrieval request to the operator and sends the user a citation (via the Citation daemon) when the message is restored. The archival of messages is done by a procedure which doesn't involve users; the user is involved only in the retrieval phase.

Intelligent terminal

The MME terminal offers many features for the application program to use in implementing a comfortable, natural interface. The terminal supports local two-dimensional editing, multiwindow capabilities, and independent memory management, which result in a natural and highly responsive interaction style. By having a terminal with powerful local editing, most of a user's word processing takes place in the terminal, thus providing predictable, timely response to keystroke activity.

Sophisticated user interface

Virtually every aspect of any interactive system has "user-interface" overtones. SIGMA's broad functionality and terminal capabilities had much to do with SIGMA's success. In addition, the user job's Command Language Processor, being natural, forgiving, and informative, played a major role in SIGMA's acceptance by the computer-naive user.

To compensate for a lack of formal training, SIGMA provides an on-line Help and Tutor facility [35]. These aids interact with the Command Language Processor while taking advantage of many of the MME terminal's features.

4.3 TENEX OPERATING SYSTEM

4.3.1 TENEX Environment--Overview

SIGMA is implemented on a Digital Equipment Corporation (DEC) PDP-10 computer running the TENEX operating system [4]. TENEX is a multiuser time sharing system with another rich set of functions imbedded as part of the operating system. User programs may access these functions through operating system calls, called *JSYS* calls. A TENEX "job" is started up when a user "logs in" from a terminal; a job may also be created by an existing job and detached to run without an associated terminal. TENEX features of particular importance to SIGMA will be described here.

4.3.2 Processes (Forks)

TENEX supports multiple interactive time-shared processes (called *forks* in TENEX) running in a paged virtual address space of 256K 36-bit words, organized into pages of 512 words. Processes may be either runnable or suspended. They are organized into "jobs" of hierarchically related forks, in which superior forks in the hierarchy may create, kill, and in other ways control one or more inferior forks.

Within a fork's virtual address space, pages may be private to the fork, or shared with pages of other forks, or even shared with pages of a file. More than one fork may have simultaneous shared access to pages of a file, or another fork address space in the same job, thus allowing for concurrent update, signaling between forks, and the like. A particular special case allows for several forks to share code from a file, thus reducing the overhead involved in page swapping.

Forks within a job may communicate by means of pseudo- (software simulated) interrupts. Forks may additionally send signals to other forks, even between jobs. Finally, superior forks monitoring inferior forks may determine the state of inferior forks and may themselves be notified of changes in state of inferiors; such state changes may serve to communicate information between forks.

SIGMA makes heavy use of the multiple-process aspect of TENEX, using separate forks within the user job to do command language interpretation, command execution, message access, folder access, and terminal interaction. Similarly, the SIGMA daemon processes also partition some of their functions among several forks within a job, as well as having several jobs, one for each class of functions. A special case of process function is used in the daemons, in which a controlling process (PC) in each daemon communicates through the TENEX file system with an operator process (CCP) to control the running of all daemon processes.

4.3.3 File System

The TENEX file system supports information transfer to and from devices attached to the TENEX system. SIGMA makes use of TENEX disk files, which are organized as collections of (possibly noncontiguous) 512-word pages, together with file description information in file descriptor blocks. Files within TENEX are named by device, directory name, file name, file extension, and version number.

4.3.4 Directories and Directory Names

A TENEX directory name specifies a particular collection of disk files, together with the file descriptor information for the files. Additionally, a particular directory name is associated with each user that logs in to TENEX (i.e., runs a job). Not every directory is necessarily associated with a user; directories may be set up as "files only," in which no user may log in under that directory's name. Such directories are used in SIGMA to hold files for messages and folders, and other files maintained by the SIGMA daemons or the SIGMA system personnel.

4.3.5 Directory Connection

When a user logs in to TENEX, the job created is both logged in under the log-in directory name and "connected" to that same directory. A fork that is connected to a particular directory, or logged in under that directory name, has owner access rights to the files in that directory. With respect to other files in other directories, the fork has different access rights, as explained under File Protection.

A fork may change the directory to which it is connected by supplying the directory and the password for the directory to a TENEX monitor call. A fork may be connected to only one directory at a time.

4.3.6 File Protection

TENEX supports a protection scheme for access to disk files in which access rights to a file are determined by the relation of the user process to the owner of the file:

1. The user process is connected to or logged in under the owning directory,

2. The user process log-in directory is in the same TENEX directory group as the owning directory,

3. Neither of the above.

The access rights granted to each of the above classes of user processes is determined for each file. The access rights distinguished for each class are read, write, execute, and append. Of these, SIGMA makes use of the read and write access protection to determine a user's access to certain data objects within SIGMA: folders, text objects, and selectors, which are requested by a user from some other owning user. These objects are accessible only through "User Directory"[15] files in the owning user's TENEX directory: If the "User Directory" files are inaccessible for reading by another user, the objects pointed to by the "User Directory" files are also inaccessible to that user. Otherwise, the objects are available for reading by that user; additionally, if a user's folder is accessible for reading, it is accessible for append access as well (i.e., citations may be appended to any accessible folder).

4.3.7 Pseudo-Interrupt System

TENEX pseudo-interrupts are asynchronous signals to a fork from other forks within a job, from terminals, or as the result of a number of unusual conditions. SIGMA uses the pseudo-interrupt system to provide interfork communication and for the detection of errors by a fork.

For communication purposes, SIGMA uses pseudo-interrupts to cause a fork to "wake up" and do some task, the details of which are in a shared area of memory or in a shared page of a file. A similar signaling mechanism is used by a fork to signal when it has finished with the task.

A fork may receive an interrupt on any of 36 interrupt channels, some of which the monitor reserves for unusual conditions, such as illegal instructions. An interrupt may occur on a channel at one of three priority levels, with interrupts at a higher priority level possibly interrupting other interrupts at lower priority levels. Interrupts on any channel may be activated or deactivated.

The reception of an interrupt on an active channel causes the interrupted process to transfer control to a specific address for the channel and to save the process program counter so that the process may resume the execution of the interrupted code. When the process dismisses the interrupt, process execution will resume at the saved program counter, unless the interrupt process changes the saved value.

For the detection of errors, SIGMA forks enable interrupts to detect most of the unusual conditions detected by TENEX, including illegal memory reference, pushdown stack overflow, and file data errors. SIGMA forks contain code to provide extensive tracing of process and job state on the occurrence of these conditions, and also provide for the saving of such states on TENEX files for later analysis.

[15] "User Directory" is a SIGMA concept referring to a list of the user's SIGMA objects (e.g., his Message Files), and should not be confused with TENEX directories.

4.3.8 TENEX Signals

TENEX provides a signaling mechanism for use among forks, possibly in different jobs. It allows a process to allocate a signal channel, to signal on a channel, to wait for a signal, and to deallocate a signal channel. TENEX provides for the strict queueing (first in first out) of signals on a channel. Additionally, TENEX supports the inclusion of one word of data with a signal if desired.

SIGMA uses the signal mechanism as a simple control mechanism between two processes in the SIGMA user job.

4.3.9 Interprocess Communication in SIGMA

Shared pages, pseudo-interrupts, and TENEX signals are all used in several schemes within SIGMA for communication among processes. These all appear in the methods found in the user job (see section 4.5, page 4-11, for details).

Within the daemon processes, various page-sharing and queueing schemes similar to the above are used. For example, Hardcopy uses shared pages to control processes printing concurrently on several printers; Archive uses a shared page to communicate with the TENEX archive retrieval program; and most daemons use the daemon queue package, a set of queues implemented in shared-access TENEX files (see section 4.13.5, page 4-102), *to communicate with user jobs and other daemons.*

Finally, some user and daemon processes monitor some attribute of various files to determine whether there is new information: the user job monitors the existence of a particular file to see if a notice is to be given out on the flash line of the user's terminal; the Reception daemon monitors the time of last write of the file of incoming AUTODIN messages to determine whether there are any messages to be processed; and the Hardcopy daemon monitors the files in its directory to be printed.

4.3.10 Multiprocess Debugger (IDDT)

IDDT is a user program invoked as necessary by a programmer to aid debugging of errand programs. Its capability of dealing with multiple processes without running within the address space of any of them is of particular value to implementing SIGMA. IDDT is not run during normal SIGMA operation.

IDDT allows the programmer to examine or change the contents of the address space of any process under it, and to monitor the execution of any process, including interrupting execution at particular places in the code ("breakpoints") so that the process may be examined and/or altered.

4.3.11 Archive

The TENEX archive system enables the orderly migration and retrieval of files not recently accessed to and from backup tape storage. The SIGMA system is carefully interfaced to the Archive system, including mechanisms to allow SIGMA users to retrieve needed files and for system personnel to cause migration of SIGMA-related TENEX files in an orderly manner.

4.4 PRODUCTION VERSUS EXPERIMENTAL VERSIONS OF SIGMA

At any one time there are two working versions of SIGMA; one is the experimental version called XSIGMA, the other is the production version called SIGMA. In XSIGMA all the tracing and error package calls written by the programmers are operative; in SIGMA these calls have been made inoperative. The effect is threefold.

1. The size of the running SIGMA system is much smaller than the XSIGMA version.

2. SIGMA is much faster than XSIGMA.

3. Errors are extremely hard to debug in SIGMA--in general, they have to be reproduced using XSIGMA before debugging is possible.

There is a semi-automatic procedure that turns the XSIGMA system into SIGMA. This procedure precedes any release to CINCPAC, where SIGMA (not XSIGMA) is run.

The same procedure applies to the daemons. Each daemon is independently runnable in either production or experimental mode. The option is made when the daemon is started by the Configuration Control Program. Production daemons can run harmoniously with experimental daemons.

4.5 USER-JOB COMMUNICATIONS

4.5.1 Introduction

Communication in the user job divides into three broad areas: data sharing, data passing, and control sequencing. The various protocols use different combinations of these methods to communicate with one another. There was no attempt to use one general method; each has evolved to meet the needs of the communicating processes involved.

4.5.2 CLP and FM Communication

The CLP and FM communicate through combinations of signals and interrupts with data passed by way of shared pages. On the one side the CLP sends a command to the FM for execution as an *Execution Request Block* (ERB) in its own shared page and then interrupts the FM on its execution channel (section 4.7.4.2, page 4-29). The ERB contains all the information necessary to execute the user's command. After the interrupt the CLP waits for a completion signal from the FM in order to resume. On the other side the FM uses the EC99 protocol (defined in ECOMM.REQ) for two purposes, to give up control after command processing and to ask for CLP services (section 4.6.2.2, page 4-15). When the FM has finished executing a command, it returns control by writing the appropriate codes and/or errors in the EC99 shared page and then signaling the CLP to retake control. When asking for CLP services the FM puts as many requests as it wants into the EC99 page and then asks for them to be processed by signaling the CLP; when the CLP is done it returns control to the waiting FM by a signal of its own.

4.5.3 CLP and FM Communication with the Terminal Driver

Both the CLP and FM communicate with the Terminal Driver through queues. They send all their dispatches to the Driver by way of a single dispatch queue, after which they interrupt the Driver for service (see section 4.7.3.5, page 4-26, for FM dispatch handling; see section 4.9.3.5, page 4-53, for Driver details). On the receiving side both the CLP and the FM have their own notice queue; the Driver knows which window the notices correspond to and who the owner of each window is. So after enqueueing the notice to the appropriate process, the Driver then interrupts it on a notice processing channel (see section 4.7.3.3, page 4-25, for FM notice handling; see section 4.9.4.4, page 4-56, for Driver details).

4.5.4 Communicating with the Access Modules

The Inter-Fork Communication Protocol (IFCP) is used for most of the communication with the access modules, MSGMOD and FACMOD (see section 4.8, page 4-37). The user program (which is either a daemon or the FM) makes its request through a subroutine-like call to the IFCP that puts the data in pages shared between the caller and the appropriate module. The access module fork is started and the results, including errors, are returned when the access module fork halts. The results are moved from the shared area as required.

Some of the data generated by certain MSGMOD calls are put into a shared area called ZT. ZT, designed to handle variable length blocks or strings, was the precursor to the TEXT package (section 4.13.2, page 4-96). Its use here is not significant; it was more efficient to use it then to convert to Text package calls. The CLP still uses ZT for some internal data handling.

4.5.5 User Job and Daemon Communication

In the user job only the access modules make daemon requests. They enqueue requests to the appropriate daemon using the standard queue package (section 4.13.5, page 4-102). There is no synchronization; errors are reported back to the user through citations.

The citation daemon is the only daemon that communicates directly with user jobs. It will enqueue citations onto the user's personal queue, if that recipient is on-line. See the description of the Citation daemon (section 4.12.6, page 4-81) and FM Alert processing (section 4.7.9, page 4-36) for details.

4.5.6 Common Data

All the forks of the user job share one page of data called *COMMON*. Each fork maps COMMON from the front-end during initialization. The cells in that page are, of course, predefined (in <IA-SIGMA>COMMON.DCL) and reflect global state information. The effect is like a FORTRAN common block with all its inherent limitations; the uncontrolled nature of how data gets into such global common areas is well known to us, yet our use of COMMON caused no serious problems.

4.6 FRONT END

The SIGMA front-end is logically five separate parts: Job Manager, Command Language Processor (CLP), Flash Line Processor, Help, and Tutor.

4.6.1 Job Management

The job management consists of setting up the SIGMA environment and controlling the log on, log off, and error-out procedures. When SIGMA starts, the main routine (DRIVER) basically does the following sequence of operations:

- Initializes the pseudo-interrupt system (see section 4.3.7, page 4-9);
- Initializes the COMMON area (see section 4.5.6, page 4-12);
- Creates the FM fork;
- Creates the Terminal Driver and starts it for its initialization;
- Initializes the CLP;
- Initializes the error package;
- Starts up the Terminal Driver;
- Terminates the job if either of the TENEX files <SIGMA-LOCAL-STATE>STOP.DAEMONS or STOP.SIGMA exists, and if so, tells the user about this condition on the MME terminal (which is why we need to wait until now to do this);
- Checks to make sure the terminal line is not locked out;
- Starts the flash processing;
- Starts the FM for initialization (see section 4.7, page 4-17, for details);
- Initializes the help system;
- Puts up the log on template and waits for the user to fill it in;
- Restarts the FM for its log on procedures (see section 4.7, page 4-17, for details);
- Initializes the user-queue for Alerts;
- Restarts the FM for Alert initialization;
- And, finally, goes into command reading mode.

As is obvious, starting up SIGMA is a complicated exercise--witness the three starts of FM, for example. It is important to realize that the sequencing is crucial, having evolved over several years.

Log out is a simpler procedure. If the user types "log off" or one of the three auto-logout conditions is met (terminal full, bad notice, terminal dead) the FM is sent a "log out" ERB to close all displayed objects and update the user's private data (see section 4.7.7.2, page 4-32, for details). Upon FM completion, an EC99 code (from the FM) tells the front-end to terminate: it closes all the appropriate files and logs out the job.

In the auto-logout cases the log out procedure is followed as described above. In the system-error case the FM is not given a chance to do a normal log out. In all log out cases the terminal is sent one line of information which describes the type of log out being executed.

4.6.2 Command Language Processor

4.6.2.1 Parsing

The Command Language Processor (CLP) reads and parses the input typed by the user and builds ERB's for the FM to process. In the case of typed commands it goes through complete parsing, produces a data collection (DCF) point (see section 4.13.6, page 4-104) when the user hits the !!*EXECUTE*!! key, and another DCF point when the FM has finished executing it. For function keys the CLP sends the FM a degenerate ERB (all the real work is in the FM for function keys) and produces a special DCF point.

If during the input phase the user wants to see the state of the parse (presumably because the CLP reported that the input is incorrect as typed), he hits the !!*PROMPT*!! key and the CLP reports to him all the commands currently possible. The list is presented in a separate window after the existing windows are temporarily unmapped. The parsing explanation below will describe what gets into this list.

The CLP parsing methods are unique within interactive systems. Most command parsers work in a left-to-right manner of fixed position parameters following the command name; each lexeme is bound as it is typed with no back-up editing allowed to bound lexemes. The CLP does its analyses on the entire line of input, thus allowing the user to edit any part of the line before parsing and freeing him from parameter order restrictions. In the jargon of natural language parsing, the CLP would be called a "semantic constituent parser" that requires the verb to be in the first position.

The CLP uses data from several sources in its parsing. Its primary one is the command table (<IA-SIGMA>COMMAND-TABLE.SOURCE) which has all the command names and the associated command bodies. Each command body describes its parameters, some data collection information, whether the command must be confirmed, a string describing the command, and the list of FM micros for executing the command. The physical position of a command within the command table defines its legal contexts--whether a file must be open for the command to be legal, for example--by using the dynamic state of the user's sessions. In addition the names of each user's personal objects (text, folders, and selectors) affect the parsing. A simplified view of the parsing method follows (for details see <SIGMA-DOCUMENTATION>FRONT-END.DOC):

1. The first lexeme of the command string is bound to all possible command name matches.

2. From the results of 1, a command option list is built, each member of which is a skeleton of a possible command and all its parameters.

3. Next the rest of the lexemes in the command string are bound to satisfy the needs of the commands in the option set. So, for example, if the user had typed

   ```
   ††DISPLAY J3N
   ```

   J3N might be bound to JAN as the month for a DTG of a message ID, J3 as the user ID for a MSGID, and J3TEXT as a text object name. Note that spelling correcting is done here. Later passes resolve the ambiguities and conflicts.

4. The basic heuristic is to pick the command which uses the most lexemes of the user's inputs and yet has the least number of unbound parameters. So the parser now discovers which command(s) uses the largest number of input lexemes and prunes away those which use less. All the commands that make it to here will get into the PROMPT display. Further pruning refines the list.

5. The CLP now checks to see if any active commands have all their parameters accounted for, i.e., no defaults are needed. If such a command is found, then all those not meeting this criteria are rejected.

6. The CLP then determines the maximum number of parameters parsed from the input and removes from consideration those commands not matching this maximum. So, for example, since

   ```
   ††FI 2,3,4 PENDING
   ```

   matches two parameters as a FILE command and only one as a FINDSTRING command (with "2,3,4 PENDING" as the search string), the FILE interpretation is chosen.

7. Now the CLP finds the command which needs the fewest parameters. Since by this point we have used the most input lexemes, this pruning leaves commands which need the fewest default parameters.

8. If there is still ambiguity at this point, the CLP computes binding strengths for all the parameters. Here for the first time it considers how much spelling correction was or was not done as a factor for choosing the best command; thus, better spelled commands which are pruned for previous reasons are not considered here.

9. If all this leaves only one command, then it is ready for execution. Otherwise, the user is given an error message saying the command is ambiguous or incomplete and a !!*PROMPT*!! suggestion is made. The commands left at this point will be highlighted in the !!*PROMPT*!! display; those rejected by previous steps will be shown but not highlighted.

The CLP attempts to make sense of almost anything the user types in the instruction window. Obviously, with the latitude the CLP allows, it is easy to "fool" and sometimes does unpredictable things. This is considered a minor annoyance when compared to the ease of use it provides to its users.

4.6.2.2 CLP services

In addition to parsing input strings, the CLP also provides services to the FM (defined in detail in the file <IA-FM>CLPAGE.BLI).

The FM can ask the user a YES/NO question by presenting it to the CLP through the EC99 protocol; the CLP then asks the user the question and puts the answer in a COMMON cell.

The CLP will also take strings from the FM and check to see if they are valid user names (typically for address list parsing). In XSIGMA the strings may be unambiguous beginning matches, while in SIGMA they must be exact matches. If they match successfully the user ID is returned for the string.

Last, the CLP will take a selector (see section 2.14.6, page 2-29) in internal format from the FM and return the TID for the string form (see section 4.13.2, page 4-96).

4.6.3 Flash Line Processor

The Flash processor is set up during SIGMA initialization as follows:

- The file which has the number of new PENDING file entries (<DAEMON-LOCAL-STATE>PENDING.DATA) for each user is mapped in.

- An interrupt channel is set up for the once-a-minute processing.

- A once-a-minute interrupt process is primed by calling the TENEX IIT JSYS.

In the once-a-minute processing the following happens:

- The flash line, consisting of the SIGMA version number, the new mail and alert counts, the load average, and the time of day, is generated and sent to the Driver.

- If the user queue is not empty, the alert processing is initiated by interrupting the FM on the alert channel.

- The next interrupt is primed for one minute later.

4.6.4 Help and Tutor

The SIGMA HELP system is invoked when the user hits the !!*HELP*!! function key: the CLP calls the help facility as a subroutine. HELP then takes control of the terminal's windows and gives the user aid by accessing an on-line database containing explanations of various SIGMA "terms." Terms are words or phrases that name particular aspects of the system. They include all command and parameter names, classes of data objects and operations, and procedural matters relevant to the user's work situation. They are essentially a set of keywords and key-phrases that semantically cover SIGMA and the environment in which it is embedded. The actual documentation for a given term consists of text called a "cell." The display of a cell can be scrolled: it is not broken into frames in the sense of screenfuls or pages.

When the HELP facility is activated the command window and work area (display window and/or view window) are mapped away, and the help information is displayed to the user in this space. (For a description of the functionality of the HELP facility see section 2.7.2, page 2-7.) When the user returns from HELP to his previous state, the information which was there before he hit !!*HELP*!! is mapped back on the screen.

The CLP interface to the facilities of the Tutor is more complex. The Tutor supports two related features: lessons and exercises. When the user executes the **LESSON** command, he must close all open objects before **LESSON** causes the text of the specified lesson to be displayed. Most of the lessons have exercises associated with them. The user may take an exercise while in a lesson by executing the **EXERCISE** instruction which takes as a parameter the number of the drill to be run. When the user enters the exercise, the working area of the screen (which had displayed the lesson) is remapped to display the exercise. When in the exercise mode of the Tutor, the user can type commands in the command window just as he would if he were not in the Tutor at all. The CLP parses these commands as always; however, the resultant parsed command is not immediately executed but is first passed to a Tutor routine for scrutiny. This routine compares the command with the list of allowed commands for this exercise. If there is a match, SIGMA is allowed to execute the command; otherwise execution is aborted. Notice that since the screening of commands by the Tutor routine is performed after the CLP has parsed them, the match is not between what the user typed and what the exercise specified but rather between the parse of what was typed and a parsed form encoded with the exercise. This means that the comparison is between the semantics of what the user typed and the desired semantics of the exercise. Any alternate form that the CLP can recognize as the same target command will pass the Tutor's inspection as the desired command. This behaves correctly in a pedagogic sense, in that the user is considered to have completed the exercise successfully if he achieves the desired result, regardless of how he achieved it.

In order to guarantee that a user can do no harm to any real data (his own or anyone else's) when doing an exercise, the Tutor must not only screen commands to be executed, but must in some cases switch the data objects on which a command operates so that "real" data is never used. If real data were used, it would also be hard for the exercise to refer to what the user was seeing when he displayed data. By using special Tutor data, the Tutor both protects the real data and has a handle on what the user sees when performing the exercise. The Tutor's only interference with the execution process is to switch data objects in the parsed command after the CLP has parsed it and before it is executed.

When the user executes a command within an exercise, SIGMA is unaware that the Tutor is involved, and the screen is used just as it would be if there were no exercise displayed. The Tutor is responsible for mapping away its own display and allowing SIGMA to display what it will. However, once the user has executed a command he must be able to return to the exercise to see what to do next. A toggle is provided which allows the user to switch the screen between the "normal" state (as it appears after executing a command) and the "exercise" state (which maps the exercise text into the work area). To prevent confusion the Tutor uses the feedback window to keep the user aware of what he is looking at and how to toggle to look at the opposite state of the screen.

For a description of the functionality of the Tutor facility from the user's point of view, see section 2.7.3, page 2-8. A more detailed description of both the Help and Tutor facilities of SIGMA is given in [35].

4.7 FUNCTIONAL MODULE

The Functional Module (FM) is the executor of the ERBs, the owner of the user's object statefiles, the manager of the Virtual Terminal (VT) for SIGMA's display and view windows, the keeper of all open objects during editing, and the focus of alert processing. While none of these individual tasks is difficult, there is complexity not only in doing them all, but in describing them as well. The distributed nature of the FM processing tasks does not lend itself well to a top-down discussion. With that caveat we start with initialization and proceed from there.

4.7.1 FM Initialization and Log On

As indicated in section 4.6.1, page 4-13, the FM initialization takes place in three steps, all synchronized with the CLP. In the first phase the following occurs:

- The COMMON area is mapped (see section 4.5.6, page 4-12).

- The pseudo-interrupt system is initialized.

- The error package is initialized for the FM (see section 4.13.1, page 4-90).

- The Driver-FM notice queue is mapped.

- The Text package is initialized (see section 4.13.2, page 4-96).

- The "PLEASE LOG ON...." message is prepared for the user.

- The SIGMA message directories are mapped (see section 4.11.1, page 4-65).

- Three of the entry point channels are set up, INTEXECUTE, INTVTDATA, and INTCLEARHERES (see section 4.7.4, page 4-28).

- The FM returns control to the CLP by halting.

With the environment set up, the CLP gets the user's log on data and restarts the FM to do the Log on:

- The VT state is set up.

- Log on is accomplished and the FM-CLP communication area is cleared. The system table entry is set up for this user (his ID, job number, security, TTY number, and whether he is running the production or experimental version of SIGMA).

- Text package identifiers are created for the EXECUTOR and TITLE names.

- Various user-specific capabilities are put in COMMON.

- System News is generated for display from a TENEX file created expressly for this purpose.

- The TEXT, FOLDER, and SELECTOR statefiles are read (see section 4.7.7, page 4-31).

- The FM returns control to the CLP by halting.

With Log on completed, the CLP maps the user's Alert queue and then restarts the FM to do its Alert initialization which includes setting up the Alert interrupt channel. Once completed the FM goes into a wait state and is prepared to be interrupted on any of its channels.

4.7.2 FM Display Management

The various displays produced by the FM are governed by a module called <u>EDIF</u>. Each display, whether it be a message for editing or a directory for viewing, has a *Source Block* associated with it. Though at most two displays may be shown at once (one in the DISPLAY window and one in the VIEW window) there may be many source blocks around for potential display. What is actually displayed depends on the state of the EDIF display list, <u>DISPLIST</u>.

4.7.2.1 Source blocks

In order to get anything in the FM windows a source block must be created. The information in a source block includes:

- the internal ID of the object;
- the logical VT window number (see section 4.7.3.2, page 4-24);
- the type of object, one of MESSAGE, FOLDER, TEXT, SELECTOR, or ALERT (source blocks created only for VIEWING have no type);
- a domain location used by the MME terminal for cursor positioning;
- a pointer to the VT display list;
- a pointer to the object's semantic structure (dependent on the type of object);
- various status information, such as security and whether the block is new or has been changed;
- object dependent data.

4.7.2.2 Source block management

The state of the FM display is determined by the contents of a structure called DISPLIST. Among other things it contains:

- a list of all the open source blocks (at most three, one each for folders, messages, and text);
- a pointer to the source block which is in (or to be put in) the Display window;
- a pointer to the source block which is in (or to be put in) the View window;
- a pointer to the Alert source block;
- a pointer to the source block which is currently being displaced by the Alert object as the VIEWed object. The !!*ALERT ON/OFF*!! function key switches the VIEWed object from this position to the VIEWed position and vice-versa.

After a command has been executed (see section 4.7.4.2, page 4-29) the routine ELMAPDISPLAYLIST does the actual work of making the MME terminal reflect the state in DISPLIST. The routine works as follows:

- Any open source blocks that are "new" to the screen get prepared for their debut by a call to the VT routine VE.INIT.
- The source block for the DISPLAY window is determined if none is currently DISPLAYed--MESSAGE, FOLDER, TEXT is the order of choice. This only happens if the last command closed the DISPLAYed object.

- The screen security is determined and written into COMMON as SECURITY.LEVEL.

- The status line, containing information about the open objects--name and security for messages and text objects; name, security, number of entries, and current entry for folders--is now generated and flashed on the terminal.

- New cursor positions are reported to the terminal.

- The VT structures are mapped on the terminal.

- Finally, the screen is updated by the VT to reflect the appropriate changes (VT.SCMUPDATE).

4.7.3 The Virtual Terminal

The *Virtual Terminal (VT)* is the concept describing the support package needed by anyone wishing to communicate with the MME terminal. The CLP and Tutor have degenerate VT's due to the limited nature of their interaction with the terminal; only the FM uses the terminal at an interaction level of enough complexity to require all the facilities about to be described.

The VT has two tasks: to manage the VT-structure for the FM and to model exactly what part of this VT-structure the terminal has in its memory. In doing these tasks it performs four services:

- processing notices from the terminal ;

- sending dispatches to the terminal;

- synchronizing the conceptual "paragraph" structure which coexists with the domain structure;

- providing the utilities needed by the FM to build and modify its VT structures.

Each of these will be discussed after a look at the VT structure itself.

4.7.3.1 VT structure--domains and paragraphs

The nature of the text which appears on an FM window is governed by its associated VT structure, a linked list of VT blocks. Each VT block in this list is created by the the FM to represent a *paragraph* of text, an arbitrary logical unit of text (of any size). As it gets to the terminal the physical representation of one VT block for each paragraph must be changed to reflect the terminal's unit of discourse, the *domain*. The domain is a numbered piece of text of limited length with specific display and formatting characteristics. So if the FM creates a VT block exceeding the maximum length, the VT must split that block into pieces satisfying the terminal domain concept, yet preserving the FM's paragraph boundaries. This duality leads to many of the complications found in the FM.[16] This paragraph-domain duality is crucial to the FM's VT operation. A VT block is six words consisting of:

- a forward and previous pointer (remember, a VT block is part of a linked list);

[16] In fact, preserving this fiction of paragraphs and domains was once considered so difficult that two separate structures were maintained by the FM, an Editor structure for paragraphs and a VT structure for domains. A streamlining of the FM late in SIGMA's development consolidated these structures

- a domain ID for terminal communication;

- an arbitrary half-word for use by the application (it usually points to the associated semantic structure, a message block or a folder entry);

- the terminal characteristics for this domain: highlight, enterable, allow HEREs, fertile (can the user hit carriage return), format (starts a line) and amount to indent;

- FM flags in this domain: new to the terminal [NEW2SCM], new to the FM [NEW2ED], in the terminal [SCMHAS], the first domain of the paragraph [FIRSTOFPARA], the last domain of the paragraph [LASTOFPARA], a new paragraph [NEWPARA], a HERE, a prompt;

- the text identifier (TID) (see section 4.13.2, page 4-96) for this domain;

- the TID for this paragraph (only present if this domain is [FIRSTOFPARA]).

The use of all this information will be shown in later parts of the document. The following example will show how the FM's paragraph conceptualization is physically realized in the terminal's domain model. Suppose the FM wants to display some text to appear as in Figure 4-4.

```
Line 1:  Now is the time for all good men to come to the aid of their party.
Why?  Who knows; people have been using this sentence for years.

Line 2:  The quick brown fox jumped over the lazy dog.  Sorry, I needed
another example.
```

Figure 4-4: A sample display

The original VT-structure for this figure as initially created by the FM would be as follows:

```
Domain Number    = 1
[FIRSTOFPARA]    = yes
[LASTOFPARA]     = yes
[FORMAT]         = yes
[HILIGHT]        = underlined
Paragraph Text   = "Line 1:"
Domain Text      = "Line 1:"

Domain Number    = 2
[FIRSTOFPARA]    = yes
[LASTOFPARA]     = yes
[FORMAT]         = no
[HILIGHT]        = regular
Paragraph Text   = " Now is ... for years."
Domain Text      = " Now is ... for years."

Domain Number    = 3
[FIRSTOFPARA]    = yes
[LASTOFPARA]     = yes
[FORMAT]         = yes
[HILIGHT]        = regular
```

```
    Paragraph Text  = " "
    Domain Text     = " "

    Domain Number   = 4
    [FIRSTOFPARA]   = yes
    [LASTOFPARA]    = yes
    [FORMAT]        = yes
    [HILIGHT]       = underlined
    Paragraph Text  = "Line 2:"
    Domain Text     = "Line 2:"

    Domain Number   = 5
    [FIRSTOFPARA]   = yes
    [LASTOFPARA]    = yes
    [FORMAT]        = no
    [HILIGHT]       = regular
    Paragraph Text  = " The quick brown ... another example."
    Domain Text     = " The quick brown ... another example."
```

There are several points to note:

- Every block is a whole paragraph (i.e., both [FIRSTOFPARA] and [LASTOFPARA]).

- Domain 3's formatted space produces a blank line in the VT structure.

- Each block has the same domain and paragraph text.

When the FM instructs the VT to put this structure out to the terminal, the structure will be modified as shown below (assuming the terminal has a domain text limit of 50 characters[17]):

```
    Domain Number   = 1
    [FIRSTOFPARA]   = yes
    [LASTOFPARA]    = yes
    [FORMAT]        = yes
    [HILIGHT]       = underlined
    Paragraph Text  = "Line 1:"
    Domain Text     = "Line 1:"

    Domain Number   = 2
    [FIRSTOFPARA]   = yes
    [LASTOFPARA]    = no
    [FORMAT]        = no
    [HILIGHT]       = regular
    Paragraph Text  = " Now is ... for years."
    Domain Text     = " Now is the time for all good men to come to the a"

    Domain Number   = 6
    [FIRSTOFPARA]   = no
    [LASTOFPARA]    = no
    [FORMAT]        = no
    [HILIGHT]       = regular
```

[17]The actual limit of the MMF terminal is 100 characters per domain

```
    Paragraph Text   = none
    Domain Text      = "id of their party.  Why?  Who knows; people have b"

    Domain Number    = 7
    [FIRSTOFPARA]    = no
    [LASTOFPARA]     = yes
    [FORMAT]         = no
    [HILIGHT]        = regular
    Paragraph Text   = none
    Domain Text      = "een using this sentence for years."

    Domain Number    = 3
    [FIRSTOFPARA]    = yes
    [LASTOFPARA]     = yes
    [FORMAT]         = yes
    [HILIGHT]        = regular
    Paragraph Text   = " "
    Domain Text      = " "

    Domain Number    = 4
    [FIRSTOFPARA]    = yes
    [LASTOFPARA]     = yes
    [FORMAT]         = yes
    [HILIGHT]        = underlined
    Paragraph Text   = "Line 2:"
    Domain Text      = "Line 2:"

    Domain Number    = 5
    [FIRSTOFPARA]    = yes
    [LASTOFPARA]     = no
    [FORMAT]         = no
    [HILIGHT]        = regular
    Paragraph Text   = " The quick brown ... another example."
    Domain Text      = " The quick brown fox jumped over the lazy dog.  So"

    Domain Number    = 8
    [FIRSTOFPARA]    = no
    [LASTOFPARA]     = yes
    [FORMAT]         = no
    [HILIGHT]        = regular
    Paragraph Text   = none
    Domain Text      = "sorry, I needed another example."
```

Again, there are some notes of interest:

- No domain text length exceeds the terminal length restriction.

- The paragraph text always appears on the block which is [FIRSTOFPARA].

- The paragraph text is the concatenation of the domain texts in the block from [FIRSTOFPARA] through [LASTOFPARA], inclusive.

- The blocks which were originally [FIRSTOFPARA] are still [FIRSTOFPARA] (naturally, they have the same domain IDs).

The isomorphism of the last point is crucial since the FM semantic structure can only point to the VT structure through this [FIRSTOFPARA] block. The VT takes great pain to keep the physical address of a [FIRSTOFPARA] block the same in case this block is deleted, because all pointers to the [FIRSTOFPARA] block are to its physical address. In other words, if the SIGMA user deletes exactly the text of the domain 2, it is not enough to just mark domain 6 as [FIRSTOFPARA] and fix its paragraph text. What actually happens is that domain 6 would be physically deleted after its updated contents were moved into domain 2.[18]

All the ramifications of this paragraph-domain duality are too numerous to mention. Suffice it to say that there are utility routines for manipulating these blocks at both the domain and paragraph level; the repercussions will appear throughout this section. Note, however, that all of this is unknown to the terminal--the FM's VT makes all the appropriate transformations.

4.7.3.2 A brief look at the terminal/VT "system"

Much of what is about to be described reflects the characteristics and terminology of the MME terminal. A full description of the terminal can be found in [39]; what follows here is a discussion of those terms relevant to the VT.

The terminal divides its logical memory into *windows*. Each window has its own associated list of domains, each comprising text and the characteristics relevant to the terminal. The window number is always present in every communication to and from the terminal.

The terminal has a *screen* on which it displays *mapped* windows. The application program controls the number of windows that appear on the screen, how many lines they take up, and where they appear. In addition to "owning" these mapped windows the terminal can also have windows which are not mapped. The terminal's memory management scheme is fairly intuitive; it gives its space first to mapped windows and then to unmapped windows. When the terminal needs space it finds it from *scrollable margins* of windows; the margins are the beginning domains and end domains of a terminal window. In the case of large objects the VT only sends out enough domains to fill the allocated window screen and a few extra domains to give the terminal some margins. If the user scrolls a window (and a former margin comes onto the screen), the terminal will send a Vacancy notice to the application program for more domains to fill the now depleted margin. When the new domains are sent, domains not on the screen are eventually garbage collected by the terminal to make room for the new domains, and these scrolled away domains are reported to the VT. The application program has a command to mark a margin as scrollable or not--it must be prudent about marking margins as nonscrollable so as not to hamper the terminal's memory management. The effect of all this is a joint effort between the terminal and the VT to put a "sliding window" over the VT structure. That sliding window has two parts: a fixed screen size part, and a variable sized margin part at both edges. When a window is unmapped, the entire structure is considered margin and subject to scrolling. So, while the SIGMA user is manipulating mapped windows, the terminal and VT are carrying out an asynchronous dialogue of Scroll and Vacancy notices to keep the screen "full" of visible text. There is nothing in this schema to prevent the VT from modifying unmapped windows (and obviously their domains) and, in fact, most of the Alert processing (see section 4.7.9, page 4-36) occurs to the unmapped Alert window.

Here is a scenario to illustrate all this machinery. Suppose the user has two objects open, a large folder which is mapped and a message which isn't. As the user scrolls down the folder, the terminal's memory fills up with the folder's domains that arrive in response to vacancy notices at the bottom margin. As its memory

[18]This is one of the pitfalls we discovered after doing the editor-VT consolidation

fills, the terminal gets new space for these new domains by scrolling away domains from the margins of the unmapped message. When the message is almost gone, it starts to cannibalize the folder at the top margin to make room for the new domains at the bottom. The user, of course, has no idea of all this. If he should now hit !!*DISPLAY OPEN MESSAGE*!!, the message, now almost devoid of displayed domains, is mapped and the folder is unmapped (all by the FM). The terminal quickly sends out vacancies for the message window and gets its space back from the now unmapped folder window. Had the margins of the folder been marked nonscrollable the terminal would die because of a "terminal full" condition.

A cursor position is another attribute maintained for each window even though the terminal has only one physical cursor. When the SIGMA user pushes the UP window or DOWN window keys to move the cursor across window boundaries, the cursor will jump to the maintained cursor position. The VT represention for the cursor position is a *descriptor*, a triple comprising a window number, a domain, and a position relative to the start of that domain.

4.7.3.3 Processing notices from the terminal--VTSV

The terminal notifies the application program of its changes and needs by way of *notices*. These notices, processed by the VT routine VT.SV (Screen to Virtual Terminal), are all at the domain level. Some of the domain-paragraph synchronization is done immediately as the notice is processed, while some happens later in VT.EDUPDATE (see section 4.7.3.4, page 4-26). The control flow of these components depends completely on the SIGMA user's actions--the section on the FM's entry points (section 4.7.4, page 4-28) will describe all the incarnations of VT.SV and other major VT activities.

There are six kinds of notices:

1. Change domain. This notice occurs when a user has typed into a domain or has deleted a domain. The changed domain is marked as [NEW2ED].

2. A domain has been marked with a HERE. When the user puts a !!*HERE*!! on the screen, the terminal highlights that character in inverse-video. Since highlights are an attribute of domains, the terminal must split the domain to effect this change. The domain is made uneditable, and since it is now a unique structure, its identity is preserved regardless of any that might go on around it. The notice the terminal sends tells the VT what it has done and the ID of the new domains it has created.

3. New domain (extract). This notice tells the VT how a new domain was created by the terminal. In the simplest case the user types enough text into a domain to overflow its length restriction; the terminal reacts by splitting the domain, sending a change domain notice for the original domain and an extract notice for the new one. There are other special cases too detailed for this discussion.

4. End-of-line. If a domain has the "fertile" characteristic, the user is allowed to hit the carriage-return key if the cursor is in the domain. Every time he does so, the terminal will generate this notice, resulting in the creation of a new formatted domain (which, incidentally, is the only case where a [NEWPARA] beginning is marked for later processing by a VT.EDUPDATE).

5. Answer vacancies. When the terminal's margin is exhausted (see section 4.7.3.2, page 4-24), a vacancy notice is sent from the terminal to the VT, which in turn responds by sending out more domains at the edge designated by the notice. A vacancy notice also indicates how many lines the terminal wants.

6. Answer scroll. When the terminal throws out domains from the margins (see section 4.7.3.2, page 4-24), it reports them through scroll notices to the VT, which in turn marks the domain as not in the terminal (not [SCMHAS]).

In all cases (except for the one pointed out in End-of-line notice) where a domain is created or deleted, the paragraph boundaries, [FIRSTOFPARA] and [LASTOFPARA], are adjusted accordingly. Only in the deletion case is the paragraph text TID modified (to eliminate the need for a special marking scheme for deleted domains). This discussion has been simplified to give a flavor of the VT actions to the six notices. A full description of all the cases within these notice types can be found in *HP/MME Terminal - Application Specification* [31]. The VT.SV routine handles them all.

4.7.3.4 Updating the paragraph structure--VT.EDUPDATE

Much of the domain-to-paragraph synchronization required by notice processing is done directly by VT.SV. The rest is done in VT.EDUPDATE.

The modified VT structures are searched for domains marked [NEW2ED] (resulting from Change domain, New domain, and End-of-line notices). If the domain is a whole paragraph (i.e., both [FIRSTOFPARA] and [LASTOFPARA]), the domain text TID is simply copied into the paragraph TID slot. This case can occur several ways; the simplest is a text edit to a short (i.e., less than the domain length limit) paragraph.

If the new domain is part of a paragraph, a new paragraph text TID is created by concatenating the domain text TIDs for the constituent pieces. The only complication occurs when a [NEWPARA] is discovered during the concatenation phase. If so, the paragraph is terminated (marked [LASTOFPARA]) at the domain preceding the one marked [NEWPARA], a new paragraph is started (by marking it [FIRSTOFPARA]), and the processing continues.

The design decision to do the paragraph-TID-adjustment in VT.EDUPDATE assumes enough localized editing so that the paragraph TID is not rebuilt several times for nothing. If the user is making single changes to many paragraphs, then nothing is gained by splitting this work from VT.SV; in fact, a needless complication is introduced. If the user is heavily modifying a document, this strategy is probably sound (though unstudied and not proven).

4.7.3.5 Getting dispatches to the terminal--VT.VS

Once the FM has prepared a VT structure for displaying, VT.VS (Virtual Terminal to Screen) puts it out to the terminal. VT.VS bases its work on a cursor position supplied by the FM. The cursor position--comprising a window number, a domain, and a character position within that domain--is the anchor from which VT.VS finds enough text above and below to fill the terminal's windows, ensuring that the cursor's domain will appear on the screen. The algorithm follows.

For each *bound* window:[19]

```
If the cursor has been specified
    then if the window is new
            then call NEWFROMCURSOR (cursor)
```

[19] A bound window is one that has been reported to the terminal. It may or may not be mapped (i.e. physically on the screen).

```
                otherwise call SHOWAROUND (cursor)
      If the cursor has not been specified
           then if the window is new
                then call NEWFROMCURSOR (first domain)
                otherwise call GTSCMHAS
```

NEWFROMCURSOR operates as follows:

1. If the window was mapped, then it is cleared.

2. The screen size (in lines) for this window is determined. Call this number X.

3. Starting at the cursor domain, the VT structure is searched upwards until (X/4)+1 number of lines are found.

4. Starting at the cursor domain, the VT structure is searched down for X+1 number of lines (note: by sending approximately 25 percent more lines than are required we provide the terminal with the margins it needs around the screen image).

5. The routine SIMPLEPUTOUT is called with the start and end domains found in steps 3 and 4.

6. The cursor position is sent to the terminal.

7. The terminal is notified that vacancies are permitted for this window.

SHOWAROUND, called when a cursor position has been specified and the structure for this window is in the terminal, tries to get text to the terminal while minimizing screen activity by taking into account what is already there.

It operates as follows:

1. If the cursor position is outside the range of what the terminal has, NEWFROMCURSOR is called and this routine exits (the effect will be to blank the window on the screen and repaint with new material surrounding the cursor domain).

2. If no change has occurred to the domains in the terminal, then the new cursor position is reported and we exit.

3. If the cursor domain is near the top or bottom edge of what is in the terminal, then the VT only updates a screensize worth of domains from that edge and deletes the rest (via PUTOUT).

   This leaves us the case where the cursor is nearer the middle (i.e., survives the "near the top or bottom edge" test). Here an additional test is made as to whether more than 20 domains have changed or been added. If so, we call NEWFROMCURSOR and return. If not, the new or changed domains that are on screen are updated (via PUTOUT).

4. The cursor position is sent to the terminal.

A lot of what happens here is "art," a euphemistic term for a special case or heuristic designed to handle any peculiarities that may appear on the screen. There are no absolute algorithms to handle these display tasks, and experimentation is difficult because of the expense in writing VT (or probably any) code.

GTSCMHAS furthers this art by doing some things that are hard to explain. In short it does the following:

1. If the text the terminal has, plus any new text inserted between those boundaries, is less than twice the number of screenlines, we call PUTOUT with the same boundaries (any new text will be inserted on the fly) and exit.

2. If the amount of text exceeds the limit mentioned in step 1, we compute new domain boundaries based on the new center of the structure between the old boundaries.

3. These new boundaries are tested for "too many changes" (as before). If there are too many, NEWFROMCURSOR is called; otherwise, PUTOUT is called.

Why isn't the "too many changes" test done in step 1 as it is in step 3? It's fairly easy to construct cases where the condition is easier to raise in step 1 when the test isn't done. Some detailed experimentation would be necessary to make sense of these "heuristics."

SIMPLEPUTOUT is the routine which cycles through the boundary domains sending out the data indicated in the VT blocks (PUTOUT does some work before calling SIMPLEPUTOUT which is beyond the level of this discussion). There are two cases:

1. The domain is in the terminal. If the text is new ([NEW2SCM] and [SCMHAS]), a "change text" dispatch is sent out. The terminal will erase the old text in the domain and replace it with the new.

2. The domain is a new one ([NEW2SCM] and NOT [SCMHAS]). A "create domain" dispatch is sent indicating the new domain ID, which domain it follows, its capabilities, format, highlights, and text.

This discussion has been much simplified, yet it conveys the level of complexity of trying to maintain a model of what the terminal has. This complexity stems from trying to make the screen as stable to the user as possible, despite the VT having limited knowledge about what is on-screen. Some of what we call anomalous screen behavior still exists because it is simply not clear which heuristics control which behavior.

4.7.4 FM Entry Points

Following initialization, the FM has four channels on which it can be interrupted for its various services. Each channel is at the same level, so that once interrupted, the FM is never reinterrupted, i.e., each interrupt runs to completion.

4.7.4.1 Processing notices from the terminal--INTVTDATA

The routine INTVTDATA is composed of a single call to VT.SV (see section 4.7.3.3, page 4-25). When it is not processing commands or Alerts, the FM is basically a VT agent for the terminal. While the SIGMA user is simply editing and scrolling, the FM is acting as a Notice handler (see section 4.7.3.2, page 4-24). Every time a notice is generated for an FM window the Driver puts it on the FM Notice Queue and interrupts the FM on this channel. When VT.SV is called, all notices in the queue are processed. As will be seen, all the FM entry point routines call VT.SV first, so VT.SV is prepared to find an empty queue.

4.7.4.2 Command processing--INTEXECUTE

Whenever the SIGMA user types a valid command or hits a valid function key, the CLP builds an Execution Request Block (ERB), puts it in the ERB page, and interrupts the FM on this channel for its execution. The FM then does the following:

1. VT.SV is called. During this interrupt the keyboard is locked, so there is no possible user activity that can result in notices.

2. VT.EDUPDATE is called to complete the paragraph/domain synchronization (see section 4.7.3.4, page 4-26).

3. Semantic interpretation (see section 4.7.6, page 4-31) on the edits processed by (1) and (2) above is done for commands which specify "pre-processing"; most commands fall into this category.

4. The command is now executed (see section 4.7.5.1, page 4-30) by calling ERBDECODE.

5. Semantic interpretation is done now for all commands which require "post-processing." Most of the text editing commands (e.g., !!*MOVE*!!, !!*PICKUP*!!) fall into this category. Without this "post-processing"[20] the user might see a screen which will be radically changed by the pre-processing of the next command. By doing it now, we give him a more consistent image.

6. The Alert object is garbage collected (see section 4.7.9, page 4-36). We do it now to avoid killing an entry which might be part of the current command.

7. We call EI.MAPDISPLAYLIST (see section 4.7.2.2, page 4-19) to update the screen.

8. Control is returned to the CLP.

4.7.4.3 Clearing HEREs--INTCLEARHERES

The life of a HERE extends for one command cycle. When the FM returns from its command execution (see the previous section, 4.7.4.2), the CLP interrupts the FM on this channel if there are any HEREs still present. The processing is as follows:

1. VT.SV is called. This call is generally superfluous, but because of a flaw in the CL.ACKNOWLEDGE process (see section 4.6.2.2, page 4-15)--the keyboard is completely unlocked, allowing the user to hit more !!*HERE*!!s with potentially disastrous effects--we do it to be safe.

2. VT.EDUPDATE is called.

3. VT.CLRHERES is called to eliminate the HEREs, by rejoining the split-off domain and resetting the highlights and domain attributes.

4. Control is passed back to the CLP.

[20] This concept was not present in early versions of SIGMA; all commands were once pre-processed.

4.7.4.4 Processing alerts--INTALERT

When the once-a-minute flash processor detects new messages for the user (see section 4.6.3, page 4-16), it interrupts the FM on this channel. The processing is as follows:

1. VT.SV is called.

2. VT.EDUPDATE is called.

3. The Alert processing routine is called (see section 4.7.9, page 4-36).

4.7.5 Executing Commands

This section will describe the processing of the Execution Request Blocks (ERB). The routine ERBDECODE, called from INTEXECUTE (see section 4.7.4.2, page 4-29), interprets the micros found in the ERB.

4.7.5.1 Micro control structure--ERBDECODE

An ERB is a sequence of micros; each micro is a SIGMA primitive responsible for some action. All user commands are defined in the SIGMA Command Table in terms of a linear execution of micros. The control structure of ERBDECODE is therefore trivial: if a micro is decoded and executes successfully, go on to the next one; if it doesn't, abort execution; when done, return to INTEXECUTE.

4.7.5.2 Executing an FM micro

Micros are physically divided into classes: messages, folders, folder entries, text objects, tutor, selectors, and general system activities. When a micro is decoded, the class, the micro number, and the parameters are determined. A CASE statement for each class finally calls the appropriate routine to do the actual work.

Each routine, besides its obvious work, is responsible for the three system actions: generating error codes, generating data collection points (see section 4.13.6, page 4-104), and checking security. Of course, the nature of these functions depends on the particular micro being executed.

There are approximately 100 micros in SIGMA. There is a fairly close correspondence between user instructions and micros. For each object type (message, file, text, entry, selector) there are a set of common operations (e.g., create, open, display, clear, delete) and a group of object unique operations (e.g., restrict files, coordinate messages, keyword entry). In addition there are a collection of system operations dealing with arbitrary objects (e.g., system news, clear view, find top, lesson, logo). To go over each one would be prohibitive.[21] Instead a few classes will be described.

Each SIGMA object has a "CREATE" micro. The processing would go something like this:

- Perform a security check to make sure the user has not specified a level higher than his maximum.

- Write a DCF point about the creation.

[21] An on-line description of each can be found in <IA-GENERAL>MICRO-CATALOG.SOURCE

- Do the creation, entering the object's name and internal ID into his directory and lexicon (this step is not done for messages).

- Write a data collection point giving the internal ID.

Displaying an object generally involves two micros:

1. OPEN the object, i.e., build a source block (see section 4.7.2.1, page 4-19). If an object of the same type is open, close it.

2. DISPLAY the newly created source block.

These micros also do security checking and write DCF points. Each micro can also generate one "error" code as part of its processing--it writes it in the FM/CLP communication page; the CLP finds the string associated with the error code and puts the message on the screen.

4.7.6 Semantic Editing

The MME terminal considers all typing uniformly as edits to domains and reports them to the application program as notices. The interpretation of these edits is wholly the responsibility of the application program. When the interpretation happens in real time we call the process *semantic editing.*

All interactive systems have one form of semantic editing, command processing. When the SIGMA user types text in the command window, that string is interpreted by the CLP as an attempt by the user to express a legal SIGMA command. A more interesting example occurs when the user edits an address list in a SIGMA message. During the command cycle following the edits, the names are checked to see if they are legal addressees. This directly prevents the user from sending a message to an unknown recipient. A similar interpretation is made of the contents of the Precedence field. Whatever the user has typed is converted into one of the four legal values for that field.

There are no other cases of semantic editing which directly affect the SIGMA user, even though others were proposed. For example, direct editing of the user's SIGMA directories, as an alternative to the DELETE or RESTORE commands, was a possibility (rejected by CINCPAC as being too confusing for a VIEWed object).

This style of editing puts more burden on both the system builder and system user, but if done properly can lead to a more natural interactive style of use. We didn't explore this area very much in SIGMA.

4.7.7 User Statefiles

Each SIGMA user has a variety of local files of personal information. Some are related to the running of his user job, one is used for data collection, one is an error log, and the rest are directories of his SIGMA objects.

Each SIGMA user has his own set of directories of objects to which he has access. Except for messages, each class of objects--folders, text, or selectors--has its directory kept in a separate TENEX file. And finally, each individual object is represented by a separate block of data in the appropriate file.

4.7.7.1 Statefiles for SIGMA objects

A folder block contains the following information: the type of block (in-order, delete, restore, new, update--each to be described in the next section), the user's name for this folder, the folder ID (system-wide), the security, the last update date/time, high-water mark information, a flag which indicates if the folder is foreign or not, (if it is foreign) the owner of the folder and from whom this user got it.

A text block contains the type of block (same as folder, above), the internal ID (local to this user), the user's name for this object, the security, the size in bytes, and the actual string.

A selector block contains the type of block (same as folder above), the internal ID (local to this user), the user's name for the selector, the security, and the actual selector.

Note the following:

- There is no user message directory. Messages "belong" to the SIGMA system, not to individuals.
- Folders are also shared objects; the statefile block contains only personal information about the folder, as well as the globally known folder ID.
- Text and selector objects are local to each user. If a user GETs one of these from another user, a private copy is made.

4.7.7.2 Maintenance of object statefiles

When a user logs on to SIGMA for the very first time, his statefiles are empty except for two built-in folders, PENDING and MYPENDING (which, by the way, cannot be deleted). This state changes by certain actions during his SIGMA session.

When he creates a new object with commands like CREATE and GET, a statefile block of type "new" is appended to the appropriate statefile. When the user has modified an existing object, e.g., executing a *!!FINISH!!* after editing, an "update" type of block is appended. In the case of the DELETE and RESTORE commands, simple "delete" and "restore" type blocks are appended. Though the exact state of the user is kept internally in the core image, these appended blocks to the old statefile represent a history of the user session to be used, as we will shortly see, in case of an abnormal termination.

When the user logs off, a new version of each statefile is written, "deleted" objects are now expunged, and the "in-order" type blocks are written out alphabetically (by user assigned names). When the user logs back on following a successful log off, the "in-order" blocks are simply read in and the user is ready to go. If the user logs on following an abnormally terminated session, the statefile processor will find various blocks following the "in-order" ones. Each one, "new," "update," "delete," or "restore," is processed so that the user's state is the same as it was prior to the crash.

The statefiles are the final word on what objects the user has access to. There is a User lexicon which the CLP uses in command parsing. This lexicon (section 4.13.4, page 4-100) keeps the correspondence between names of objects and internal IDs, but it can get out of synchronization with the statefiles (e.g., during a system crash). A utility named UFILES can rebuild the lexicon from the statefiles should this problem exist. Of course, we could rebuild the lexicon each time during log on to avoid the synchronization problem, but that would add a considerable delay to that process.

4.7.8 Internal Handling of SIGMA Objects

4.7.8.1 Messages

When a user asks to display (or view) a message, the FM gets the information necessary to build the display from MSGMOD (see section 4.8, page 4-37). The information is organized by fields of the message, and one block (called a *main message* block) is created for each field (or field paragraph for multiparagraphed fields). These main message (MM) blocks form a linked list in the FM. Each MM block contains the field type, a TID for the text of the field, and other information. The building of the VT structure is dependent on field type. The FM cycles through the MM blocks and builds an appropriate list of VT blocks for each MM block depending on its type. The MM block has pointers to the first and last VT blocks corresponding to it, and the first VT block points back to its MM block. The form of the display is also dependent on the type of the message. There are three message classes (AUTODIN, MEMO, and NOTE) and each can be of preparation or released form, giving a total of six different formats.

When a user edits a message, the internal structure (i.e., the list of MM blocks) must be synchronized with the VT structure. To do this the FM cycles through the fields of the message and creates, deletes, or modifies MM blocks as necessary so that they correspond to the current VT structure for each field. This is done during each command cycle.

In addition a semantic analysis of user editing is done. This analysis is field dependent. For address fields the names supplied are broken into separate domains and checked to ensure they refer to valid user identifications. For those that don't, an asterisk is inserted in the display after the name. For multiparagraph fields the user may request (by hitting the !!*UPDATE*!! key) that formatting be done. If he does, text is rearranged as necessary to eliminate extra blank space within paragraphs. This requires the FM to modify both the VT structure and corresponding MM blocks appropriately. When the message is closed, its current state, as represented by its list of MM blocks, is reported to MSGMOD.

Comments may also be attached to any field or field paragraph, and any string of text within an editable paragraph can be highlighted (i.e., displayed in inverse video). The appropriate VT structure is created, and an MM block is added for each comment and highlight. For highlights the MM blocks contain the character positions of the beginning and end of the highlighted string. For both comments and highlights the access codes (private, public, for a specified user) are stored there as well. As comments or highlights are modified or deleted, both the VT structure and associated MM blocks are changed appropriately.

It should be noted that a user may have a message in the view window as well as one "displayed." In this case, since MSGMOD can handle only one message at a time, a second MSGMOD fork is created to handle the "viewed" message.

4.7.8.2 Folders

When a user first enters a command requiring access to a folder, the FM issues a request to FACMOD (see section 4.8, page 4-37) to get all the entries for the folder in question. FACMOD builds an array of five word blocks where each block contains all the information for a single entry (including TIDs for actual text). This array is then mapped in by the FM. This form of internal storage facilitates access to entry information as the user moves around within a folder.

Folders are handled differently from other objects because of their size. In general, folders are too large to keep in core. So actual VT structure is initially built for only twenty entries (at most) around the "current" entry. Each entry block in the array has pointers to the beginning and end of its corresponding VT structure, and the VT structure contains the associated entry number.

As the user scrolls or otherwise moves through a folder, VT structure is built for additional entries. If the new entries referred to are sufficiently close to those for which VT structure exists, all entries in between are also built so that the VT structure always represents a contiguous list of entries. If the new entries built are not close to those in the existing VT structure, the structure for the old entries is removed from the linked list but is still kept around. Only when more than 100 entries have VT structure are VT blocks actually deleted. That is, VT structure is reserved in case it is needed again until the space required for it gets excessive. When structure has to be removed, first unconnected structure is removed. If there is none or if more structure must be deleted, that which is farthest away from the entries currently on the screen is removed. The rationale for not implementing a **VIEW FILE** instruction is that it would require a second FACMOD fork dedicated to the folder just to allow scrolling through it.

There is an internal table of state information (FILEGLOB) kept for the currently open folder. Among other things this table contains pointers to the first and last entries in the display list, the current entry, the open entry (if any), the number of entries in the folder, the number for which VT structure exists, the number selected in a restrict, the security of the folder, and a pointer to a linked list of folder blocks containing state information for all folders to which the user has access.

If a user asks to see only a selected portion of a folder using the **RESTRICT** and **AUGMENT** commands, FACMOD is called to get the list of entries satisfying the specified selector. If the user is doing a restrict within a restrict, FACMOD is also passed a list of entries in the current display since those are the only ones to be considered in the next restrict. FACMOD returns a list of entries which satisfy the restrict or augment criteria, and the FM then removes the current VT structure and builds the appropriate new one. Each time a restrict (or augment) is done, the list of entries corresponding to the currently displayed subset is stacked (along with the "current" entry associated with that subset) so that its display can be reconstructed if the user "backs up" to it via !!*BACKUP ONE*!!.

When a user specifies a folder entry in a command, that entry is "opened." That is, a call to FACMOD is made to get the message ID for that entry and the ID is written in COMMON. That entry becomes the "current" entry and the status line (see section 4.7.2.2, page 4-19) is changed to show it as such. On the display the current entry is distinguished from the others by showing its entry number with normal video and underlined. (All other entries have their entry numbers in inverse video.)

SIGMA also maintains a *high water mark* for each user for each folder he has access to. This keeps track of the last entry the user has seen in the folder. The high water mark is used for positioning the folder when the user displays it. That is, it becomes the current entry. Only the **EMPTY** and **SORT** commands reset this high water mark.

Often a user will wish to see just his recent entries. When the user restricts RECENT he sees only those entries which have been added to the folder since he last displayed it. Notice that this is not necessarily all entries beyond the high water mark since a user may have displayed a folder sometime without looking at all his recent entries.

The user may modify the entries themselves in several ways. Entries may be highlighted (i.e., the subject line displayed in inverse video) and comments may be attached to them. Existing comments and highlights can be removed or modified by their creator. The highlight and comment information is sent to FACMOD where it is maintained along with the other entry information. The FM gets this information back from FACMOD as required for building the display, and, for each entry, maintains it internally as a linked list of blocks attached to the associated entry block. The same representation is used for both comments and highlights. A flag indicates what kind of block it is. Changes to highlights and comments (including access specification changes) are made in the FM's internal representation and eventually reported to FACMOD. FACMOD is only told about changes as part of the semantic interpretation cycle (which happens during normal command processing) or whenever the block for an entry is about to be removed. If a user removes all the text from a comment, then the comment is deleted (i.e., FACMOD is told to mark it deleted) as soon as its associated entry block is removed.

Users can also delete entries from folders they own. However, entries are only marked as deleted at the time the delete command is executed. Only when the folder is closed are entries actually removed. If entries which have been marked deleted are shown in a display (e.g., the user says, "AUGMENT DELETED") they are shown in half intensity.

If a user closes a folder which he does not own, and if he has made only changes to personal data (e.g., changed his high water mark), a complete folder update is not required. FACMOD is told to abort the folder, and the FM does a <u>fast</u> <u>folder</u> <u>update</u> by recording all changes in the folder statefile. This is done strictly for efficiency.

When a new folder is to be created, a new temporary FACMOD fork is created in which a folder ID is obtained and a folder is built; both these pieces of information are passed to the Folder daemon for registration in the folder directories. This sequence is done in order to make the new folder ID available to the user job. If the Folder daemon were to totally create the folder, it would have to pass back the ID to the user job somehow. Note that this process creates a window during which time the user thinks the folder exists (following command execution), but it isn't accessible because the Folder daemon hasn't registered it yet in the folder directory. Notice that this FACMOD fork not only makes the new ID available but creates the actual folder as well, thus shortening the window length mentioned above. The FACMOD fork is killed at the end of the create command.

4.7.8.3 Text objects

All the information about the user's text objects is stored in his text object statefile (see section 4.7.7, page 4-31). When he logs on, the file is read, and a directory containing the names and the first 20 characters of each text object is built. The complete text objects themselves are read in only when needed. The text object directory information is stored in a linked list of blocks (kept in alphabetical order) from which the actual VT structure for individual text objects is built. This information includes the text object name, its security, its length, a TID for the first 20 characters of text, and a TID for the entire text. The state of the currently open text object (if any) plus a pointer to the linked list of text information blocks is kept in an internal table within the FM.

If a user creates a new text object, an information block is created for it and inserted alphabetically into the linked list, and the corresponding VT structure is built. Unlike folders, if a user GETs another's text object, a private copy is created for him. If a text object gets changed, its information block is appropriately modified. If the user requests that formatting be done (by hitting the !!*UPDATE*!! key), text is rearranged as necessary to eliminate extra blank space within paragraphs. This requires the FM to modify both the VT structure and the associated text object information block. If a user deletes a text object, it is just marked as deleted until he logs off or creates a new text object with the same name, at which time the text object is actually removed from the system. Text objects marked for deletion appear with asterisks next to their names in the directory when displayed. On log off the updated information for all text objects is written into the u 's statefile.

4.7.8.4 Selectors

A directory of selectors is built for the user upon log on by scanning the information kept in his selector statefile (see section 4.7.7, page 4-31). The selectors are stored in alphabetical order in a linked list of blocks (one per selector). This information includes the selector name, its security, and a pointer to the actual selector. An internal table keeps state information about the currently open selector and a pointer to the linked list of blocks containing the information about all the user's selectors.

If a user creates a new selector, a selector block is created for it and inserted alphabetically into the linked list. As with text objects, when a user GETs a selector from another user, a private copy is created for him. Unlike text objects, however, selectors are not editable. Hence, to change a selector the user must create a new selector with the same name. The user may request that a selector be deleted, but as with text objects it is just marked as deleted until log off or until the same name is used again in the creation of a selector. The directory will show each selector marked for deletion with an asterisk next to its name. When the user logs off, the updated selector information is written into his statefile.

4.7.9 Alerts

When a user logs on to SIGMA, an alert queue (see section 4.13.5, page 4-102) is established and the alert object source block (see section 4.7.2.1, page 4-19) is initialized. An initial structure is created consisting of two VT blocks, one a header saying "ALERT LIST" and the other a trailer saying "END OF ALERT LIST." If the user has an alert selector defined (i.e., a selector with the name "ALERT_SELECTOR"), a pointer to it is attached to the alert object.

As a user's incoming citations are processed by the Citation daemon, they are placed in his alert queue if he is logged on. Each time the flash line is updated (i.e., once a minute, see section 4.6.3, page 4-16) the alert queue is checked to see if any citations have come in. If so, an interrupt is sent to the Functional Module (FM) on the alert channel (see section 4.7.4.4, page 4-30). If the user has his pending file open, surrogate entries are built and dynamically inserted at the end of his pending file. (These entries appear identical to the real entries which are added to the user's pending file by the Citation daemon.) Whether or not his pending file is open, the citations are checked to see if they meet the alert selector criteria. If they do, entries are built and entered into the alert object, and the terminal bell is rung. Note that these are fat citations (see section 4.12.6.2, page 4-81), so they can be quickly processed.

The alert object source block points to two linked lists of blocks. The first contains the VT structure for the display, and the second contains the message IDs, precedence, and security for each entry, as well as pointers into the VT structure.

After the alert object has been updated, COMMON is changed to show the number of alerts now in the alert object and, if the user's pending file is open, to show the number of entries which have been added to it since it was opened.

An arbitrary limit of ten entries has been established for the alert object. During the normal command cycle the number of entries in the alert list is checked, and if it exceeds ten, all but the last ten are removed. Since this garbage collection occurs only following the execution of a command, the alert list could get arbitrarily large between command executions.

The alert list is shown to the user in the view window. Although it may share this physical window with another object, it has its own logical window. The !!*ALERT ON/OFF*!! key causes the logical alert window to be mapped into the view window if it is not there and to be mapped out if it is. If another object is sharing the view window, that object gets mapped out when the alerts are mapped in and vice versa.

When the alert list is in the view window, the user may display the associated messages by placing a HERE in the entry and hitting the *!!DISPLAY ENTRY!!* key. The user obtains the ID of the message for the requested entry by using the back pointer from the VT block (that corresponds to the location of the HERE) to its associated message ID block.

4.8 THE ACCESS MODULES

The Access Modules perform the manipulation and management of SIGMA messages and folders (files). Existing as separate processes and serving both the SIGMA user job and the background daemons, the Access Modules provide a uniform mechanism for the access, modification, and updating of these complex objects.

4.8.1 Rationale

While TENEX provides a virtual address space for its application programs, the limits of that virtual space can easily be reached when sophisticated programs that manipulate large data objects are run. We realized early in our design of SIGMA that messages and folders could become very large, as would the code which used them, and that it was not acceptable to constrain them both to the TENEX virtual address size (256,000 words).

We examined two alternatives to solve this problem. The first was to segment the SIGMA objects into TENEX pages, swapping them in and out of the virtual memory as needed. This technique was used to advantage in some earlier systems, notably the On-Line System (NLS) pioneered at SRI. Its primary advantage is its ability to handle objects of arbitrary size. Our analysis showed it to have several significant disadvantages for our application, however:

- Performing our own page swapping would have introduced a second level of paging overhead, in addition to that imposed by TENEX itself. For the types of data references that we needed, we predicted that this overhead could be disastrous for SIGMA's performance.

- The orientation around TENEX pages would have required a data organization which fit within those 512-word units. Earlier efforts such as NLS suffered from that shoehorning; we did not wish to constrain our data formats to such arbitrarily-sized units.

- Paging-oriented systems require the awareness of the paged nature of the data to be distributed throughout the code which manipulates it. Given the high degree of change we expected in SIGMA as we refined it, we did not wish to carry the burden of page management along.

The second alternative we considered was to place the code which managed a large data object in a separate address space along with the data. Assuming that the code which directly manipulated the data could be kept fairly small, this would significantly increase the amount of virtual memory available for the data without imposing any organizational (e.g., page) constraints. Although this approach could not handle arbitrarily large objects, we determined that each of our objects would be able to fit into an address space using such a scheme, and we could reference as many large objects as we needed by creating more address spaces.

The multiple address space organization exhibited none of the above weaknesses of the paging scheme, although it would introduce some unique problems of its own. We finally decided to implement it, resulting in the programs known as the Access Modules: *MSGMOD* for messages, and *FACMOD* for folders.

The Access Modules contain the code which directly operates on message and folder data, taking up part of a separate address space. Sharing that address space with the object it manipulates, this code is responsible for passing requested data to other processes for their use and modifying data within the object upon demand. In addition, each Access Module performs services for its callers which are more efficient by virtue of its closeness to the object.

4.8.2 Access Modules: Brothers Under the Skin

Although they manipulate different objects, the Access Modules have many attributes in common. Many of these stem from the fact that they were designed to meet address space limitations, but their continued development caused them to grow together in other aspects as well.

Inter-fork protocols

Because the Access Modules "live" in a separate address space from the callers which actually use the data (for display, etc.), they cannot be called through standard subroutine linkages. Instead, Access Module routines are called through the IFCP mechanism[22] (see section 4.5.4, page 4-12).

Many masters

Messages and folders are manipulated both in the foreground by the SIGMA user jobs and in the background by the various daemons. Since the Access Modules are responsible for the physical management of these objects, they are required to perform in both of these capacities. They, alone among the SIGMA processes, had to conform to both types of running environments.

Daemon gateway for user job

The Access Modules issue all calls to daemons to initiate background processing. This insulates the rest of SIGMA from needing to know the specific formats for daemon requests.

Delta-file oriented

The multiaccess nature of messages and folders was a significant design consideration for the SIGMA system [43]. To permit parallel access and update of these objects by many users, the Access Modules employ a *delta-file* technique to record changes made by users. This technique allows changes made in parallel to be expressed consistently, preserving the intent of the users and the integrity of the objects.

As a part of this scheme a user who has just updated a message or personal folder must be prevented from accessing that object until after the appropriate daemon has completed the update. The user is locked out from the object when the user job access module turns on a *busy bit* while making a daemon update request. There is one busy bit per user for messages, located in the message itself, and just one busy bit per folder, for the owner, located in the folder directory (see section 4.11.2, page 4-69). Whenever a user attempts to access a message or folder, the access module first tests the state of the busy bit. These busy bits are reset by the appropriate daemon when it has completed the file update.

[22]Exceptions are the direct calling of MSGMOD by the Reception and Citation daemons; see section 4.8.3.2

4.8.3 MSGMOD: The Message Module

The Message Module is a specialized package for the manipulation of SIGMA messages. Acting on directives from its callers (the FM and the Message, Citation, and Reception daemons), MSGMOD creates, modifies, and updates messages, and performs several other utility services.

4.8.3.1 Structure of SIGMA messages

A message in SIGMA is a more complex entity than in other systems. A Preparation message can contain one or more *versions*, where a version contains a set of values for each of the message fields. Versions are created by several users, each making different changes to the message's contents, resulting in several slightly (or perhaps greatly) differing renditions.

A *Transmitted message*, not subject to significant revision since its contents are of record, contains only one version; the only changes allowed are the addition or modification of comments. An *In-Preparation* message can be seen and modified by many coordinators during the drafting process. Each of these users wants to see what some or all of the other coordinators have done with the message, either by comment or modification, while wanting to be able to perform such operations himself. To allow this, each coordinator is given his own version of a message, called a *messagette*. It contains his current values for all message fields and any comments he has made. He may change any fields in his messagette, using the text editing features of the MME Terminal or SIGMA. In addition, a coordinator can view the versions of other coordinators by using the **VIEW VERSION** command, and, if he wishes, copy selected passages from those versions with the !!*COPY TEXT*!! function. In any case, changes made by a user are local only to *his* messagette; users may not change messagettes belonging to other users.

The message drafter creates the message and the first messagette. A coordinator's messagette is initialized to the value of his delegator's version, since that is what the delegator wants him to critique or comment upon. As the coordinator edits, his messagette diverges from this initial content and thereafter contains his current version.

4.8.3.1.1 Overall message structure

The format of a particular message depends on whether it is an In-Preparation or Transmitted message, and whether it is contained within a disk file or being referenced in virtual core. Figure 4-5 shows the various parts of a message and in which contexts they appear.

- The *messagette directory* contains the information describing the overall attributes of the message: its message-ID, security level, and type.

- For each user designated as a coordinator (see section 2.13.2, page 2-21) in an In-Preparation message, the messagette directory has a *user descriptor*, which describes his state: his delegator and signoff status, and a pointer to his messagette.

- The *messagette storage* area contains the messagettes, in packed format, for each of the coordinators in an In-Preparation message.

- Since messagettes can grow in size when they are modified, they are not modified in the storage area, lest they overflow into the neighboring messagette. When modification to a messagette is required, it is placed in the *current messagette* area (*CMSGETTE*). This area, as large as the maximum size of a messagette, is used to make all changes. When the changes are complete, the messagette is rewritten into the storage area, with reshuffling of the other messagettes if its size has changed. The CMSGETTE area also serves as the single messagette for a Transmitted message.

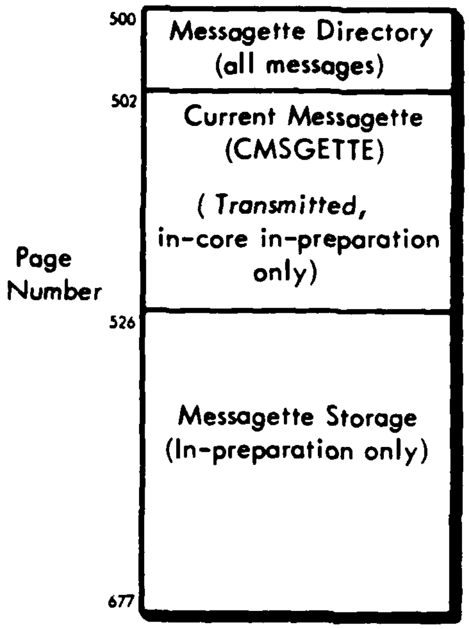

500
Messagette Directory (all messages)
502
Current Messagette (CMSGETTE)
(Transmitted, in-core in-preparation only)
Page Number
526
Messagette Storage (In-preparation only)
677

Figure 4-5: Overall structure of a SIGMA message

While running with a message loaded, MSGMOD's environment has the format depicted in Figure 4-6. The MSGMOD code occupies the lowest part of the address space. Above the code and below the message space is a buffer area, containing dynamic buffers, queues, and directories. Directly above the message space is a communication area, called ZT, used to pass various data to the FM. The IFCP communication pages are used by the IFCP package to effect the necessary communication between MSGMOD and its caller (see section 4.5.4, page 4-12).

4.8.3.1.2 Messagette structure

As described above, a messagette contains a complete version of a message, either the single version of a Transmitted message or one coordinator's version in an In-Preparation message. Figure 4-7 shows the structure of a messagette.

The *messagette header* stores overhead information, including the owner of the messagette, editing history, chop status, and memory allocation information.

The rest of the messagette's contents is stored in the *message fields*, each field corresponding to one of the defined message field types (e.g., **TO**, **SUBJECT**, **BODY**). A *field descriptor* is reserved in the messagette header for each defined message field, pointing to a linked list of one or more *message field blocks*. The field may have one of two forms:

1. The field may be *indirect*, in which case the value of a user's field is defined to be whatever value another user has. In this case the field is represented by an *indirect field block*, which names the user whose field describes what is to be used. This type of specification is useful when initial values of messagettes are created during the coordination process.

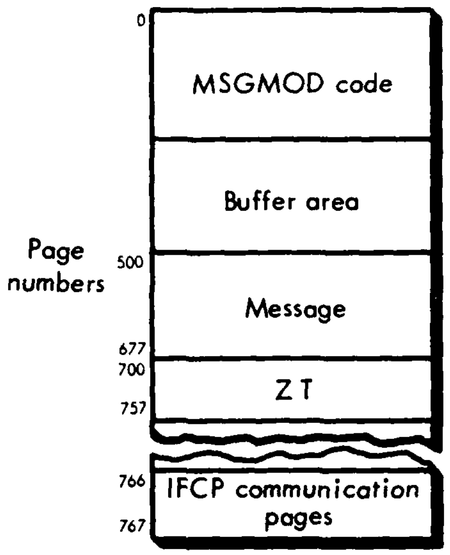


Figure 4-6: The MSGMOD running environment

2. The field can have normal format, in which case it contains a list of one or more *normal field blocks*. Each block can be a *datum block* which contains one datum of the field's value (e.g., an addressee or a paragraph), or a *comment block*. Comment blocks can contain either the text of a comment or a specification of a portion of a field to be highlighted, and can be arbitrarily interspersed between datum blocks.

All message field blocks are the same size, and are allocated from the messagette's space by a fixed-block allocation scheme.

It must be noted here that since messagettes can move around freely within the virtual address space, all address references within a messagette are relative.

4.8.3.2 Direct-loaded MSGMOD

In two instances, the callers of MSGMOD do not need its full generality to perform their tasks. The Citation daemon (see section 4.12.6, page 4-81) has no need to modify a message; it merely has to retrieve the contents of certain message fields to create citations. The Reception daemon (see section 4.12.7, page 4-82) creates SIGMA Transmitted messages from AUTODIN messages received from LDMX, and needs but a handful of MSGMOD routines to accomplish that task. In both these cases, the daemons need only a small fraction of MSGMOD's calls, and, because of the large volume of traffic handled by each, would be slowed considerably by the IFCP overhead.

To handle these special cases, MSGMOD can also run in a direct-loaded form; the applicable MSGMOD code is loaded directly into the load module for the daemon. The daemon routines call the MSGMOD

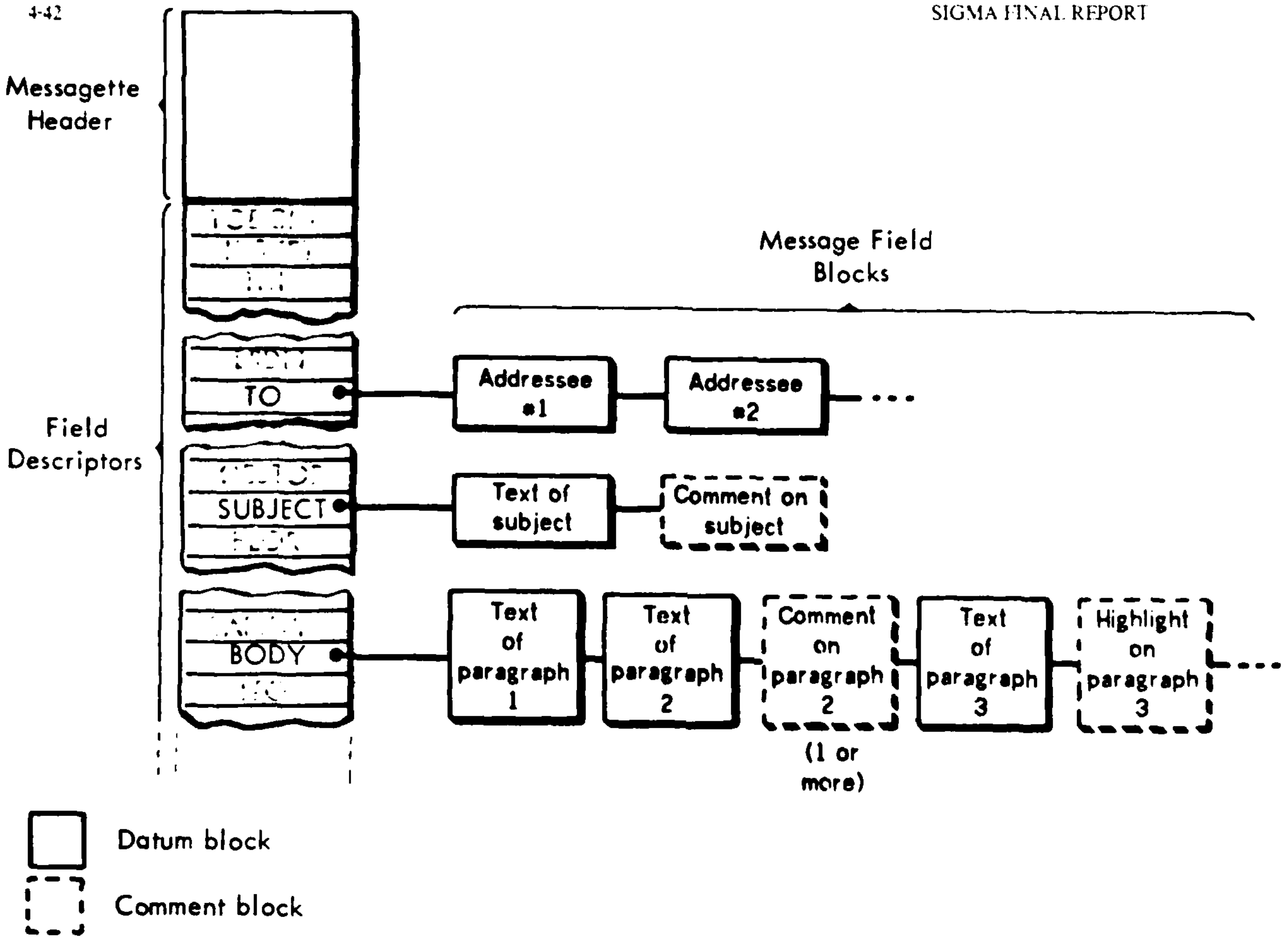


Figure 4-7: Messagette structure

routines directly via subroutine linkage instead of through the IFCP. In each case, the daemon code and memory allocation must conform to the MSGMOD environment picture shown in Figure 4-6.

4.8.3.3 MSGMOD functions

The following gives a brief synopsis of the functions supported by MSGMOD.

User Job I/O support:

- create a new message
- read in an existing message from the message database
- update a modified message (generating a delta-file)
- request *For_Action* and *For_Info* citations to be sent to users specified in ACTION, FORWARD, and ROUTE commands
- request retrieval of archived messages through the Archive daemon (see section 4.12.8, page 4-85)

User Job coordination support:

- specify coordinators and releasers and request *For_Chop* and *For_Release* citations
- get the coordination status of a specified user
- record the chop status for a user

Daemon I/O support:

- create a Transmitted message
- read/write specified message files
- incorporate a delta-file generated by a User Job MSGMOD
- prepare and send Fat Citations (see section 4.12.6.2, page 4-81)
- transform a message from In-Preparation to Transmitted form

Message field functions:

- add, modify, or delete a message field entry
- retrieve the contents of a message field
- create or expand an indirect specification (pointer to another user's field)
- retrieve or modify the contents of a messagette header

Miscellaneous:

- initialize and terminate MSGMOD
- clear (abort) a message already in the MSGMOD buffer
- identify message daemon function (preparation or transmission)

4.8.4 FACMOD: The Folder Module

Similar to the way MSGMOD handles messages, FACMOD performs analogous functions to handle folders for its callers (the FM, and Folder and Citation daemons). The term *folder* is the internal name for the object known to a SIGMA user as a *file*. The folder object is used to represent SIGMA's *Pending Files*, *Date Files*, *Action Logs*, *readboards*, and *personal files*.

Folders in SIGMA are used to store and manipulate *citations*, short abstracts which describe message transactions. Each citation, which when stored in a folder becomes a *folder entry*, allows a user to retrieve the referenced message, assign user-specified keywords, and perform sophisticated searching to locate messages satisfying certain criteria.

4.8.4.1 Structure of SIGMA folders

Figure 4-8 shows the overall structure of a folder.

- The *folder header* contains the pertinent overhead data for the folder, including its security level, owner, and storage allocation information.

- The *data block* areas store the data describing folder entries, comments, and users of the folder. The *large data block* area and *small data block* area are separate, fixed-size block areas which store the two different sized blocks used in folders. When expanding to accommodate more blocks, these areas grow toward the center, i.e., the large block area grows toward larger addresses and the small block area grows toward smaller addresses. This allows the boundaries between the segments to change without having to relocate the addresses of allocated blocks.

- The *inversion lexicon* is a SIGMA lexicon (see section 4.13.4, page 4-100) which stores the current pairings between user-specified keyword strings and the folder entries with which they are associated.

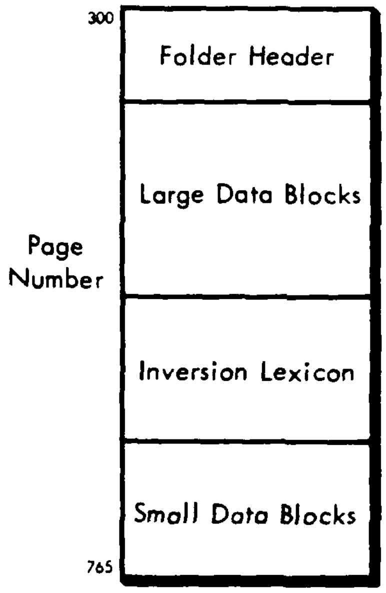


Figure 4-8: Structure of a SIGMA folder

The nonheader areas are given initial memory allocations based on an approximation of their expected maximum size. Should any one of these areas exceed its initial allocation, the space assigned to other areas is reduced and more space is assigned to the growing area, until all available space in the folder is consumed. This typically permits storage in excess of 5000 folder entries. When FACMOD is running with a folder loaded, its running environment resembles that of MSGMOD as shown in Figure 4-6.

The logical structure of a folder is shown in Figure 4-9. The folder entries are represented as a linked list of *folder entry blocks* (large), pointed to by an anchor pointer in the folder header. Each folder entry block contains an abstract of data describing a single message transaction. Should additional information be required beyond the capacity of the folder entry block, the entry can have one or more *supplemental data blocks* (small) to store the additional data (currently no supplemental blocks are needed). Comments on a folder entry are kept in *comment blocks* (small), maintained on a separate linked list anchored to the entry.

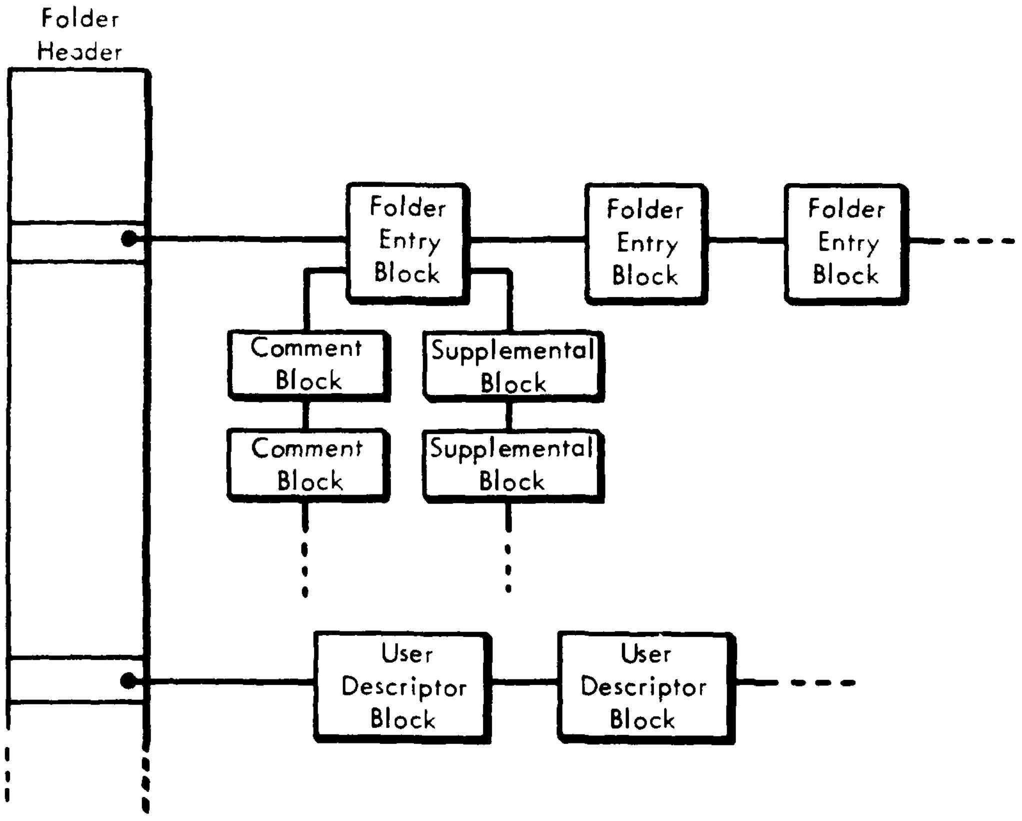


Figure 4-9: A SIGMA folder: The logical view

For each user allowed access to a folder, SIGMA maintains specific information, such as the part of the folder the user has seen and which entries have been entered into the folder since the user last saw it. This data is kept in *user descriptor blocks* (large), one per user accessing the folder, and is anchored by a pointer in the folder header.

4.8.4.2 A highly tuned facility

FACMOD's initial implementation was based on a fairly simple model of folder operations. As folders evolved and their operations grew more complex, efficiency became an important factor, and many operations were reconfigured or reimplemented to place the processing burden most effectively:

- The manner in which data about the folder entries was passed to the FM was greatly streamlined to permit responsive handling of large folders. The abstract of data about each entry was reduced to the minimum size necessary for folder display.

- Special calls were added to allow the FM to modify entries in limited ways without requiring the full generality of the modification scheme.

- The application of SIGMA selectors to folder entries was implemented entirely in FACMOD, since it had access to all the relevant entry data.

- Special code was added to FACMOD to support the *fast folder update* facility (see section 4.7.8.2, page 4-33), which allowed certain kinds of folder modifications to be performed without going through the normal expensive folder update procedure.

- A special concept called a *surrogate entry* was added. This allowed FACMOD to enter "place-holder" entries, which could then be manipulated, deleted, and selected like normal entries, without being actually added to the folder. This was used to implement the Alert and on-line folder update facilities (see section 4.7.9, page 4-36).

4.8.4.3 FACMOD functions

The following gives a brief synopsis of the functions supported by FACMOD.

User Job I/O support:

- create a new folder

- read in a folder from the folder database

- update a modified folder (generating a delta-file)

- append a list of entries to a destination folder (generating a delta-file)

- request deletion of a folder by the Folder daemon

Daemon I/O support:

- create a new pending folder

- read/write a folder file

- incorporate a delta-file generated by a User Job FACMOD

Folder entry manipulation:

- build an abbreviated FM structure to represent the entries in a folder

- add a new entry to a folder

- add a surrogate entry to a folder

- retrieve the full data associated with a folder entry

- modify all or part of a folder entry
- mark an entry (or list of entries) deleted or undeleted

Comment and supplemental data manipulation:

- add a new comment (supplemental)
- retrieve all or part of comment (supplemental) data
- modify comment (supplemental) data
- delete supplemental data

Folder entry selection:

- restrict the entry list according to a selector
- augment the entry list according to a selector

Keyword lexicon support:

- add new keyword/entry correspondences
- retrieve entries associated with a specified keyword
- retrieve keywords associated with a specified entry
- retrieve all keywords associated with a folder

User descriptors:

- retrieve user descriptor data for a user
- modify user descriptor data for a user
- set the recent status for a user

Miscellaneous:

- initialize and terminate FACMOD
- change the security filter for an open folder
- clear (abort) a folder already in the FACMOD buffer
- empty a folder
- sort a folder

4.9 THE TERMINAL DRIVER

The Terminal Driver is the function within SIGMA responsible for communication with the MME Terminal. Running asynchronously from the rest of SIGMA, it responds to the various I/O requests initiated by the Terminal and SIGMA during the terminal session.

From a functional standpoint, the Driver transmits blocks of data between SIGMA and the Terminal. Those sent from SIGMA to the Terminal are called *dispatches*; those from the Terminal to SIGMA are called *notices*. Although the reliable delivery of these blocks is the Driver's primary responsibility, it also sends and receives data and control information which effects a robust transmission protocol.

4.9.1 Rationale

The Driver is physically implemented as a pair of processes, a *Transmitter* and a *Receiver*, running asynchronously with respect to each other and the rest of SIGMA. This implementation seems somewhat extreme compared with the analogous function in other interactive systems, but was necessitated by several important considerations:

- *Isolation from the MME Terminal.* The MME Terminal supports a functional model which is radically different from most terminals (for an overview of the MME Terminal, see [39]; for a functional description, see [31]). Because many of the communication requirements are artifacts of the MME Terminal's implementation and were subject to significant change during the Terminal development, the decision was made to shield most of the SIGMA job from the lowest level details of the communication dialogue. Thus, outside the Driver, SIGMA is unaware of such details as the format of Terminal transmissions, the internal codes used to specify Terminal operations, and so forth.

- *Implementation of the Transmission Protocol.* SIGMA communicates with its Terminals through a complex array of communication gear, including multiplexors, crypto units, asynchronous line interfaces, and a front-end processor. The resulting reliability of terminal I/O is well below 100 percent. Because SIGMA and the Terminal rely so heavily on perfect synchronization and accurate data, the inherent unreliability of the transmission hardware is improved by the MME Terminal Transmission Protocol (described in [31]), providing for transmission validation and explicit acknowledgment. If a transmission error is found, SIGMA and the Terminal immediately switch to a resynchronization dialogue, determining the last of the recent transmissions which was correctly received. Any transmissions following the verified one are then retransmitted.

- *Asynchronous Operation.* The division of responsibility between the Terminal and SIGMA results in an asynchronous dialogue, wherein either side may issue communication at unpredictable intervals. The asynchronous implementation of the Driver allows it to carry on this dialogue without greatly complicating the rest of SIGMA with such issues as polling.

4.9.2 Overall Driver Structure

Several common elements bind the two processes of the Driver. These provide a superstructure which includes facilities for initialization, access to utility functions, and synchronization.

Share buffer

In order to facilitate communication between the two forks of the Driver, a segment of each fork's address space is shared between them, allowing either fork read- or write-access to any data in that segment. The shared segment is divided into two parts: the Transmission Buffer (described below) and the shared state variable area. The shared state variable area contains those variables which are of interest to both forks of the Driver, such as state indicators and shared flags.

Inter-Fork interface

The Inter-Fork Interface is a set of routines which provide necessary initialization procedures and the mechanisms to allow the two forks of the Driver to communicate with each other. Using the Share Buffer and various TENEX interrupt facilities, the Inter-Fork Interface permits the forks to signal, interrupt, and synchronize with one another.

Common binary code

The two forks of the Driver use the same binary core image (TENEX.SAV file). In addition to providing identical services for the use of the Transmitter and Receiver, this sharing permits the Driver to economize on total code space in the running system.

4.9.3 The Transmitter

The Transmitter is the output side of the Driver, and is responsible for sending any data destined for the Terminal. The input to the Transmitter comes in the form of dispatches issued by the various other processing parts of SIGMA which communicate with the Terminal. It is the responsibility of the Transmitter to accept these dispatches, convert them into a form suitable for transmission, and send them. In addition, the Transmitter acts as the output channel for the Receiver when the Receiver needs to send any Protocol signals. A block diagram of the Transmitter, showing its data structures, components and neighbors, is shown in Figure 4-10.

4.9.3.1 The Transmission Buffer--TBUF

The Transmission Buffer (TBUF) is the central data structure of the Transmitter, containing all *transmissions* waiting to be sent or acknowledged. A transmission is the form in which a dispatch is transmitted to the Terminal. This involves:

- Adding the appropriate header and trailer

- Quoting any characters in the body of the dispatch which are not allowed in the Transmission Protocol (control characters)

- Computing and appending the checksum

The retransmission nature of the Protocol allows for transmissions to be sent repeatedly until successfully received. Because of the significant amount of computation involved in producing a transmission from a dispatch, each transmission is computed once and stored in the TBUF. When the transmission is due to be sent, the literal string stored in the TBUF is sent; no further computation is necessary, resulting in efficient handling of retransmission.

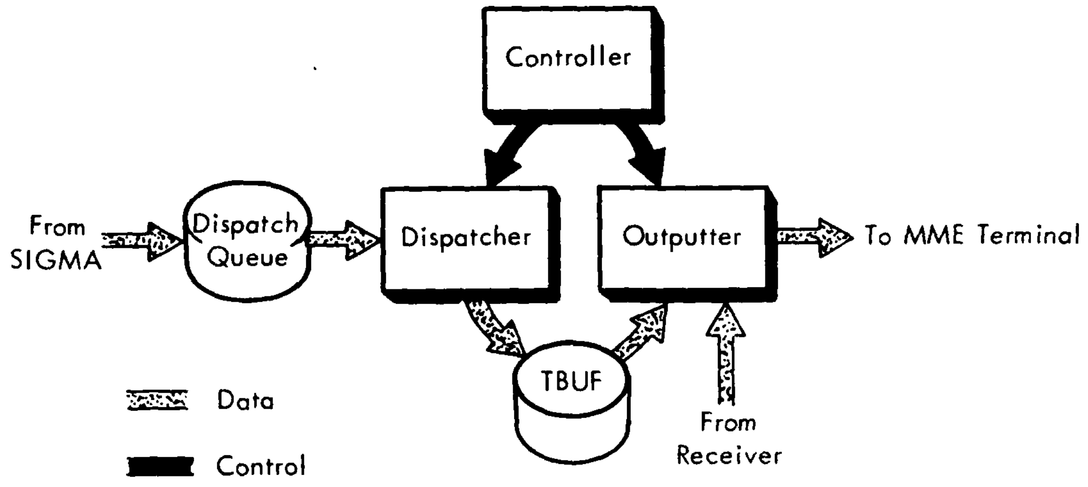


Figure 4-10: The Transmitter

The Protocol calls for each transmission to be circularly numbered in the range from 100 to 177 octal. This allows for a maximum of 64 transmissions to be simultaneously "in the pipe," i.e., awaiting transmission or acknowledgment. As each transmission is acknowledged, its corresponding entry in the TBUF is released, allowing a new transmission to occupy that slot.

The format of a TBUF entry is shown in Figure 4-11. The entry has a header which contains various descriptive information about the transmission, as well as the text of the full transmission itself:

Timestamp	The internal TENEX date/time at which this transmission was sent (0 implies that the transmission has not yet been sent). This is used to determine if an acknowledgment to this transmission is overdue.
Overall length	The overall length of the transmission string, including header and trailer, quoting characters and checksum.
Checksum length	The number of characters used for the checksum sequence. This is necessary if the transmission must be renumbered during a RESET sequence (see section 4.9.4.3, page 4-55), in which case the checksum must be recomputed and a new sequence appended.
Checksum value	The current twos complement of the sum of the characters in the transmission. Should a recomputation of the checksum be required, only the characters changed need be added and subtracted from this value; the entire sum need not be recomputed.

```
+---------------------------------------------------------------+
!                                                               !
!                          Timestamp                            !
!                                                               !
+-----------------------------+--------+------------------------+
!                             !Checksum!                        !
!        Overall length       ! length !     Checksum value     !
!                             !        !                        !
+-----------------------------+--------+------------------------+
!                                                               !
!                       Start of checksum                       !
!                                                               !
+---------------------------------------------------------------+
!                                                               !
!                      End of transmission                      !
!                                                               !
+-----------+------------+--------------------------------------+
!           !Transmission!                                      !
!    STX    !   number   !                                      !
!           !            !                                      !
+-----------+------------+                                      !
!                                                               !
!                                                               !
!                    Body of quoted dispatch                    !
!                                                               !
!                                                               !
!                                     +----------+--------------+
!                                     !          !   Checksum   !
!                                     !   ETX    !   sequence   !
!                                     !          !              !
+-------------------------------------+----------+--------------+
!                                                               !
!          Extra space for appending control sequences          !
!                                                               !
+-------------------------------------+----------+--------------+
```

Figure 4-11: Structure of a TBUF entry

Start of checksum The byte pointer to the beginning of a checksum sequence. This indicates where a new checksum should be placed if it is recomputed.

End of transmission
The byte pointer to the end of this transmission. Used by the transmitter to append Protocol control characters when necessary.

The rest of the entry contains the transmission itself, in the following order: the *STX* (start-of-transmission) ASCII control character; the transmission number for this transmission (range 100-177 octal); the body of the dispatch, with all control characters quoted into the noncontrol range; the *ETX* (end-of-transmission) ASCII control character; the checksum sequence. Following the transmission itself, there is extra space in the entry where Protocol control sequences can be appended (see section 4.9.3.4, below).

4.9.3.2 The controller

The control for the Transmitter is handled by an interrupt-driven program called the Controller. Each possible action to be taken by the Transmitter is initiated by a TENEX pseudo-interrupt on a specific interrupt channel. These actions include:

Start	Begin (or resume) transmission, beginning at a transmission number selected by the Acknowledgment Processor (see section 4.9.4.3, page 4-55).
Protocol	Send out one or more Protocol signals as specified by the Receiver.
Reset	Reset the Transmitter's state, as part of the handling of the catastrophic Protocol Reset sequence (see section 4.9.4.3, page 4-55).
Dispatch	Process a waiting dispatch. This basically entails looping on calls to the two principal modules in the Transmitter, the Dispatcher, and the Outputter, until the *Dispatch Queue* is empty and all the data in the TBUF has been sent. These are exception conditions, which are described below.
Purge	Perform a text-package purge of a text file prior to the closing of an object.
Wakeup	Attempt again to enqueue new dispatches after the TBUF was found full (see Dispatcher, below). This is a different entry point to the basic loop initiated by Dispatch.

4.9.3.3 The dispatcher

The Dispatcher supplies the Transmitter with its dispatches to be sent to the Terminal. It takes entries from the *Dispatch Queue* (see section 4.9.3.5, page 4-53), performs the necessary conversion into transmission format, and stores the resulting transmission in the next free slot in the TBUF. If there is no entry waiting in the Dispatch Queue, then the Dispatcher does nothing. If there are more entries but the TBUF is full, the Dispatcher signals the Controller that there was no room, which causes the Controller to set a wakeup interrupt for itself to attempt to enter the entries later.

4.9.3.4 The outputter

The Outputter performs all actual output to the Terminal, through the specially-designed SOUT2 (String Out) JSYS. The output consists of two different kinds of data:

1. Outgoing transmissions, coming from the TBUF.

2. Protocol signals, issued by the Receiver (R/FSM and AK/FSM).

The single Outputter function handles both types of output because Protocol signals must be contiguous, and if output were allowed to emanate from more than one source, TENEX could not guarantee that the two different streams would not get intermixed.

The Outputter is invoked on almost every activation of the Controller. If sending transmissions is allowed (during normal, error-free periods), the Outputter selects the next transmission due to be sent for output. If there are any Protocol signals waiting to be sent, the Outputter appends the character string for th› signal to the end of the transmission. This allows the Outputter to send the entire sequence with one SOUT2 call, minimizing TENEX overhead. If there are no pending transmissions or if sending transmissions is inhibited (during error-recovery periods), the Outputter sends any waiting Protocol signals by themselves.

4.9.3.5 Communication with SIGMA

Communication between SIGMA and the Transmitter is effected through the TISUB package. Each SIGMA process which can generate dispatches causes them to be sent to the Transmitter by calling a routine in TISUB with a similar name, e.g., to send a *Flash* dispatch, the caller would invoke TI.FLASH. Each of the TISUB interface routines takes as parameters the various data required for the actual dispatch, and packages them into a specially formatted block. The interface routine enqueues the block onto the *Dispatch Queue*, a multiwriter/single-reader queue, and then interrupts the Transmitter on the *Dispatch* pseudo-interrupt channel, which notifies the Transmitter that more dispatches are available.

4.9.4 The Receiver

The Receiver is responsible for reading all input characters from the Terminal, determining what the input means, and performing the appropriate actions. This task is complicated by the fact that the Terminal generates two different kinds of output: Protocol control, used to determine the state of the Terminal Protocol, and notices, which are the results of substantive actions performed by the Terminal. The Receiver performs these two different functions by acting as two different kinds of processors, as depicted by Figure 4-12.

The Receiver accomplishes its tasks with several finite-state-machines (FSMs), each attending to a different part of the Receiver's function. The following sections describe the functions of each of the FSMs.

4.9.4.1 The input multiplexer--IM/FSM

Characters coming from the Terminal are read by the Input Multiplexer. The IM/FSM must separate the Protocol signals from the notices, and route them to their appropriate processors. The IM/FSM's job is fairly simple. Although the Terminal can intermix the multicharacter Protocol signals with notices, Protocol signals are always contiguous and have rigid formats. And since Protocol signals have only a limited number of initial characters, the IM/FSM looks for any of the known starting characters. A character not in this set is processed as part of a notice. If a Protocol signal initial character is read (none of which can legally appear anywhere in a notice), then the IM/FSM enters one of its finite states to read the rest of the sequence. If the next incoming character(s) are in the proper format for the Protocol signal, then the signal is encoded into internal form and passed to the appropriate FSM. If the sequence does not conform to the required format, the budding signal is ignored and an "invalid data" code is sent to the R/FSM (see below).

The IM/FSM also monitors a timer, which is used to time various events. Should the timer ever "go off," its triggering is encoded as a special internal Protocol signal ("Timeout") and passed to the AK/FSM for handling (see section 4.9.4.3, page 4-55).

4.9.4.2 Notice reading: Quote and reception processing

Notices in transmission form contain a large amount of overhead data in addition to the relevant information they are carrying (see section 4.9.3.1, page 4-51). The notice-reading process parses the incoming transmissions, breaks them into their component parts, verifies the significant fields, and packages successfully received notices for routing to the appropriate part of SIGMA.

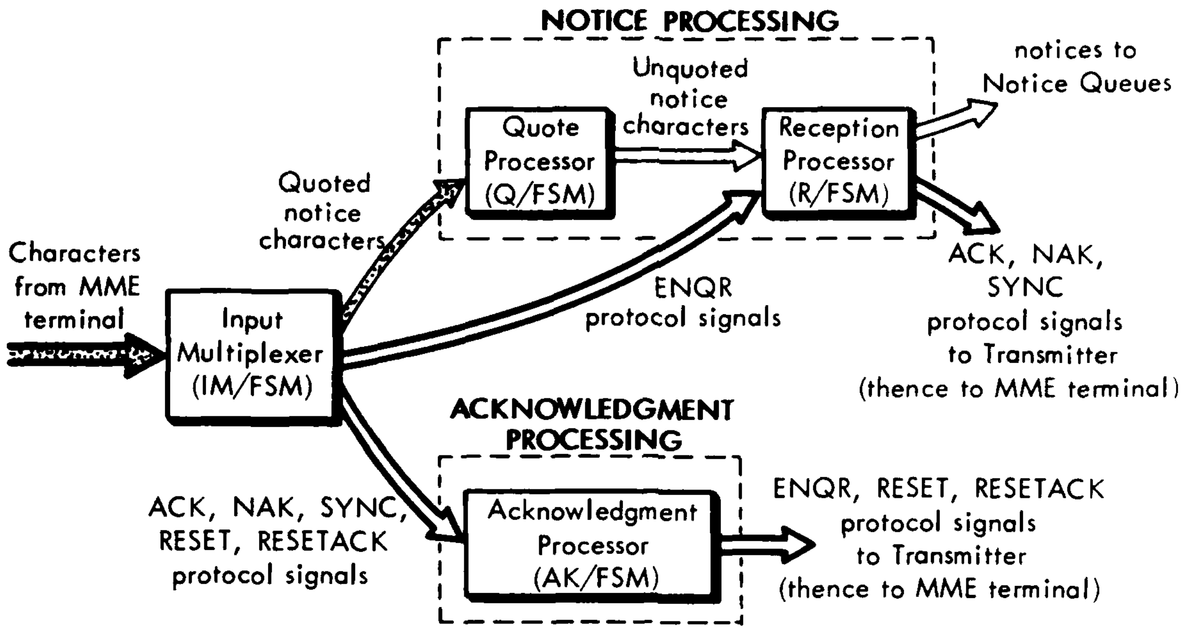

NOTICE PROCESSING
Unquoted notice characters
Quote Processor (Q/FSM)
Reception Processor (R/FSM)
notices to Notice Queues
Quoted notice characters
Characters from MME terminal
Input Multiplexer (IM/FSM)
ENQR protocol signals
ACK, NAK, SYNC protocol signals to Transmitter (thence to MME terminal)
ACKNOWLEDGMENT PROCESSING
ACK, NAK, SYNC, RESET, RESETACK protocol signals
Acknowledgment Processor (AK/FSM)
ENQR, RESET, RESETACK protocol signals to Transmitter (thence to MME terminal)

Figure 4-12: The Receiver: Two logical processors

The notice reading is handled by two FSMs, the Reception Processor (R/FSM) and the Quote Processor (Q/FSM). The R/FSM controls the parsing of the transmission; the Q/FSM is an auxiliary processor for the R/FSM which recognizes quote sequences and transforms them into their internal form.

Transmissions are parsed in the following sequence, each step of which corresponds to a finite state in the R/FSM:

1. The R/FSM looks for the *STX* character which must start every transmission. All other characters are flushed.

2. The transmission number is read. If it is out of legal range or not the next in sequence, the transmission is in error.

3. The next character, *n*, the first character of the notice proper, is the length of the notice body. If it is out of range for legal notice lengths, the transmission is in error.

4. The R/FSM reads in the next *n*-1 characters (as determined above) which become the body of the notice.

5. The *ETX* character is expected; if not found, the transmission is in error.

6. The terminating checksum is read. If this value is not the twos complement of the running sum of the characters in the transmission, the transmission has been garbled somewhere.

If, during this process, any flaw is found while processing a transmission (including an "illegal data" signal or a character not legal within a transmission, the R/FSM sends a *Negative Acknowledgment* (**NAK**) Protocol signal to the Terminal. This signal informs the Terminal that there was a transmission error of some kind, tells the Terminal the transmission number of the last successfully received notice, and requests retransmission of any notices sent since the successful one. If all of the steps are satisfied, a valid notice has been received. A *Positive Acknowledgment* (**ACK**) is sent to the Terminal to indicate successful receipt, and the notice is sent off to its SIGMA destination (see section 4.9.4.4, page 4-56).

The R/FSM keeps track of the last successfully received transmission number. This capability allows it to respond to a confused Terminal's request for resynchronization. The **ENQR** Protocol signal is thus routed to the R/FSM, which responds with the last correctly received transmission number.

4.9.4.3 The acknowledgment processor--AK/FSM

The primary responsibility for implementing the Protocol rests with the AK/FSM, the most complex of the Receiver's FSMs. The AK/FSM maintains the state of all transmissions already sent, responds to positive (**ACK** and **SYNC**) and negative (**NAK**) acknowledgments, handles catastrophic resynchronization requests (**RESET**) and senses overdue events ("Timeout"). In performing these functions, it acts as a throttle control for the Transmitter, determining which transmissions are to be sent and when it is appropriate to send them.

In the simplest case, when no transmission errors occur, the AK/FSM simply records receipt of positive acknowledgments from the Terminal. Each time a transmission is acknowledged, its slot in the TBUF is released (see section 4.9.3.1, page 4-49), providing room for another transmission to be enqueued.

Special action is initiated by the AK/FSM when a break in the normal sequence of acknowledgments occurs. This can be caused by a **NAK**, a timeout on an overdue acknowledgment, or an **ACK** specifying a transmission number which is not due to be acknowledged, any of which implies a problem in transmission. In all of these cases, the AK/FSM attempts to get back in step with the Terminal by means of the *enquiry* dialogue.

When a transmission problem is encountered, the AK/FSM first disables the Transmitter, to prevent further transmissions from being sent until the transmission problem has been resolved. It then initiates the enquiry dialogue by means of the *Enquire* (**ENQR**) Protocol signal, and sets a timer to indicate when a certain amount of time has elapsed. The **ENQR** signal requests the Terminal to reply by returning an ultra-reliable *Synchronize* (**SYNC**) signal identifying the number of the last transmission it successfully received. If the Terminal replies within the time limit with a valid number, the AK/FSM updates its "last acknowledged" counter and the Transmitter resumes transmitting at the next transmission number.

If the Terminal does not respond within the time limit or returns a **SYNC** with an inappropriate transmission number, then a catastrophic breakdown in communication has occurred. Although the situation is severe in such a case, the AK/FSM makes one last attempt at reestablishing communication with the Terminal by sending the *Protocol Reset* (**RESET**) signal. This instructs the Terminal to drop what it is doing, reinitialize its Protocol state, and reply with the *Protocol Reset Acknowledge* (**RESETACK**) signal. If the Terminal so replies, the AK/FSM resets its own Protocol state, instructs the Transmitter to renumber any pending transmissions to the initial counter value (hence the reason for keeping such detailed information about the checksums; see section 4.9.3.1, page 4-49), and instructs the Transmitter to resume. If the Terminal does not reply, it is assumed to be dead and a notice is sent to the CLP requesting an auto-logout (see section 4.6.1, page 4-13).

4.9.4.4 Communication with SIGMA

Once a notice has been received, the Receiver must send it to its appropriate recipient within SIGMA, either the Front-end (CLP, Prompt, Help, or Tutor) or the FM. This determination is made by examining the notice's type and window (if any): notices with a window specified are routed by examining the owner of the window; notices without a window (error notices) are all sent to the Front-end.

Once the appropriate recipient has been established, the Receiver enqueues it on one of the two *Notice Queues*, one each for the Front-end and FM. These notice queues have a single-writer/single-reader discipline. Once the notice has been enqueued, the recipient is notified by means of a TENEX pseudo-interrupt that the new notice is available.

4.10 HP/MME TERMINAL

4.10.1 General

From the earliest days of the IA project, the terminal to be used has been considered an important part of the system. It is the cutting edge of the user interface. The characteristics it presents the user very much determine the user's perception of the service behind it. For this reason a development effort went into designing a terminal especially for MME which would reflect the functional performance we desired. To make it as maintainable as possible this terminal was based on a standard commercial product.

The terminal chosen was the Hewlett-Packard (HP) 2649 with minor physical modifications and special firmware developed by ISI. The physical modifications were primarily to support 8 security lights, used for direct feedback to the user about the security of the objects he is dealing with (see section 2.10, page 2-13).

The ISI report *HP/MME Terminal - Application Specification* [31] is a complete description of the black box characteristics of the terminal. The communications protocol between the terminal and the host computer is described, including the communication line (transport) protocol.

A general discussion of the motivation behind the MME terminal and a functional description of it are contained in a paper presented at the 1979 National Computer Conference [39]. The following description is extracted from that paper to provide the necessary context for the subsequent discussion.

4.10.2 Functional Description

Since the HP 2649 terminal is microprogrammable, the functional operation is entirely defined by the microcode. Communication between the application program in the host computer and the terminal is done in blocks of data, representing a complete command from the application to the terminal (dispatch) or a complete report of some new condition in the terminal to the application (notice). The ISI report *HP/MME Terminal - Application Specification* [31] gives the format of all legal dispatches and notices.

The terminal is basically a half-duplex device, and at any time it is either in input state (keyboard active) or output state (host computer active). During input state the user has at his disposal the full screen editing capability. The terminal switches to output state whenever a function key is pushed. (The EXECUTE key, which causes SIGMA to interpret and execute the contents of the Instruction window, is a function key.) During output state the keyboard is disabled. The terminal is returned to input state by the host sending a special "Continue" dispatch. Strictly speaking, the system is not half duplex because in output state the terminal may send certain control notices required to maintain consistency between the terminal's database and the host's model.

4.10.2.1 Windows

The MME terminal is designed to hold up to seven separate items of text in what we call "windows." Windows are allocated and deallocated by the host (never by the terminal). They are of arbitrary length so long as the total contents of all the allocated windows does not exceed the memory capacity of the terminal. Although the host fills the windows by sending data to them, the terminal does its own memory management and decides what data to keep when it nears its memory limits.

Windows may be thought of as numbered text buffer areas. A window may be assigned by the host to occupy any contiguous portion (full horizontal lines) of the screen, such as lines 15 through 23, with an operation we call "map." Normally a window contains more lines of text than will fit on the mapped area (it may also contain less). "Map" places that portion of the data that will fit from the window onto the screen, while any excess data beyond the screen area is stored in "margins," areas logically considered to be above and below the window's mapped screen area. Data scrolls on screen from these margins.

Several windows may be mapped on different areas of the screen at the same time. A window may also be unmapped, which means it remains in the terminal memory, but is not visible to the user. The host may switch the contents of the screen from one text item to another very quickly by "unmapping" the displayed object and mapping the new object. Mapped windows scroll independently. The !!*ROLL UP*!! and !!*ROLL DOWN*!! keys cause scrolling in whatever window the cursor is in.

4.10.2.2 Domains

In the MME terminal all text is stored in "domains." Each window is made up of a contiguous string of domains. Domains are the atomic units for the communication of text between the host and the terminal; they may be any length up to 100 characters. Any character stored in the terminal can be uniquely identified by its window, domain identifier within the window, and its character position within the domain. Domains have format, highlight, and control attributes which are assigned at the time the domain is created. The user is not aware of the domain structure of text, except as domain attributes are apparent to him.

Normally each domain starts at the character to the right of the last character of the previous domain. A domain may wrap around onto the next line. However the application program may set a domain to be "formatted," which makes the domain start on the next line at the left margin of the screen, regardless of where the previous domain ends. In this case the blank space to the right of the previous domain is essentially undefined to the host, since it cannot be identified by domain ID and character position. The terminal will not allow the cursor to move to an undefined location. If a user attempts to move his cursor into such an area, it will jump to the next enterable domain.

The HP 2649 allows any combination of blinking, underlining, inverse video, and half brightness on a character by character basis. The MME terminal limits this highlighting to a domain basis; that is, all characters in a domain are highlighted the same. In addition, character set selection is done as a domain attribute.

Domains also have editing control attributes which the host sets when domains are created. The attributes control whether the cursor may enter the domain, whether the domain is editable (from the keyboard), whether characters within the domain may be marked with a HERE, and whether the domain will accept carriage returns. Space is left for assigning other attributes.

When a user edits text he is changing the contents of some domain. If, for instance, he inserts or deletes characters, the domain expands or contracts appropriately and the domain is recorded as "Changed." Nothing is sent to the host until the user begins to edit another domain. The terminal will then send the host the new contents of the previously edited domain (via a Changed Domain notice), and record this new domain as the Changed domain. Thus the host computer may be, at most, one domain change behind what is in the terminal. Eventually the user will push a function key, carriage return, or the !!*HERE*!! key. The first action the terminal takes on these keys is to send out the pending Changed Domain notice if there is one; it then locks the keyboard. Thus the host always has the up-to-date state of the terminal before it performs any operation on the data. (All SIGMA commands are initiated by function keys.)

Most often data is displayed from the application program, in which case the host computer creates the domains and sends them to the terminal. However, the terminal will generate domains in three special instances.

1. When user editing causes a domain to exceed 100 characters, the terminal will generate a new domain and tell the host its location, ID, and contents in the form of a special notice called an "Extraction" notice.

2. A carriage return creates a new "formatted" domain and a special EOL (end of line) notice is sent to the host, reporting the location of the inserted space and the ID of the new domain. Note the normal ASCII character for carriage return is not stored in the text.

3. The !!*HERE*!! key marks the character at the cursor position by making it a new, one-character domain with inverted video and noneditable attributes. This mark will stay associated with that character regardless of any editing the user might do before the command is executed. Just reporting the character position in the domain and the domain ID is not sufficient. The HERE notice contains all the information needed to identify the position of the marked character, how the old domain was split, and the IDs of the new domains created.

4.10.2.3 Flash lines

Part of the user interface to the message service involves keeping the user informed of the status of the system. This information is strictly output only, and it is not necessary to keep a corresponding data structure for it in the host. To simplify this form of output, the terminal has an eighth window called the Flash window. If it exists at all, it is assigned to the top of the visible screen, and can be set by the application to occupy from 0 to 24 lines. It has no domain structure and has fixed attributes. It is always not-enterable, and has no highlight or formatting properties.

4.10.2.4 Cursor control

Although there is just one visible cursor, each window may have an implicit cursor position. When more than one window is mapped on screen the !!*UP WINDOW*!! or !!*DOWN WINDOW*!! key causes the cursor to move to the implicit cursor position of the adjacent window.

The host has two controls over the implicit cursor for a window: 1) on what character in the window it should be, and 2) on what line on the screen (for mapped windows) it should be. For unmapped windows, this latter translates to "on what line on the screen it would be if the window were mapped." A separate dispatch is provided for each. This limited form of cursor control is the only way the host can establish what data is shown on screen. Since the user can scroll the screen contents, this is transient control at best. The host also has a dispatch to put the actual visible cursor into the desired mapped window (at the implicit cursor position).

4.10.2.5 Scrolling

The terminal's basic heuristic for mapped windows is to keep extra lines of text in margins above and below the lines that are on-screen. This lets the user scroll in either direction without having to go to the host for more data. Whenever the terminal assesses that a margin is getting too small, it will send a Vacancy notice to the host asking for more data for that margin. The terminal calculates the number of lines of data to ask for based on the size of the on screen area, the number of lines in the margin, and the amount of memory left in the terminal.

4.10.2.6 Memory management

The 12K bytes of display memory in the terminal are allocated as necessary for each dispatch. When the remaining memory is reduced to a prescribed limit, the terminal tries to reclaim memory from unmapped windows and, if that does not yield enough, from large margins of mapped windows. Memory is reclaimed by deleting domains and their contents from the edge of the margin and then telling the host through a Scroll notice. A Scroll notice identifies the last domain deleted and from which margin it came. It is important that the terminal be able to generate Scroll notices even when the keyboard is locked and the host is in control. It is during this state that the host will send the data that reaches the memory limit. The terminal must be free to reclaim memory right away in order to have room for the next dispatch, which may already be in the terminal's input buffer.

It is possible for the terminal to try to reclaim memory from the same margin the host is writing into. To prevent this the host can set special "No Reclaim" controls for each margin of each window. The terminal will not reclaim memory from a window margin so marked. The host must be careful not to leave these No Reclaim controls on, otherwise the terminal will quickly run out of margins from which it can reclaim memory, and the terminal memory will become full.

4.10.2.7 Communication discipline

Communication between the terminal and the host is really one computer talking to another. Each transmission must be error-free; otherwise the computer's model and the terminal's model may not match. To ensure the needed reliability of data across potentially noisy lines, a fully synchronized block retransmission protocol is used.

4.10.3 System Design

The firmware for the HP/MME terminal resides in Programmable Read Only Memory (PROM). This allows occasional changes to the code (requiring erasing the PROM and burning in new code) yet the terminal acts as though it has ROM memory. 32K bytes of memory are allocated for the firmware. 12K bytes are allocated for display list storage, and 8K bytes are for additional data (domains and window data and program variables).

The 2649 contains standard off-the-shelf boards from HP, except for the firmware memory boards. HP did not offer a PROM memory board at the time this program began, so ISI developed its own memory board that was compatible with the HP terminal. Each board holds up to 16K bytes of PROM memory (2708s). Two boards are required for the MME terminal. Because of high power consumption of the 2708, the PROM board has a special circuit which activates power only on the set of 8 memory chips being addressed. Power switches in less than 100 nanoseconds.

Originally much of the HP/MME firmware was written in PLM, a high-level programming language for the Intel 8080. However, the compiler was very inefficient with memory, and the firmware had to be converted entirely to assembly code to make it fit. To minimize the number of PROMs that had to be reprogrammed when a change was made to the firmware, the firmware was broken into 32 1K byte segments and directly allocated to specific PROMs. The code was organized into four levels. The lowest level contained what was effectively the operating system. The next level consisted of common subroutines that performed fundamental functions. The intermediate level contained more complex, less common operations. The highest level contained the main control loop. Calls from higher levels to lower levels (calls never go the other direction) are made through a table of entry vectors at the beginning of each level. This way, when a lower level routine is changed, only the PROMs that hold the changed routine and the vector table are changed. The higher level routines that issue calls to the changed routine do not change. This reduces the number of PROMs affected when code changes are made.

One of the most difficult problems with the terminal has been checkout of the PROMs when they are reprogrammed. We found that the PROMs would fail in the field after they had tested successfully in the lab. By testing the PROMs at elevated temperature (50 degrees Centigrade), we were able to expose marginal 2708s and greatly reduce the incidence of failure in the field. A part of the lowest level firmware is a Self-Test feature which runs a sum check over all of PROM memory as well as pattern testing RAM memory. We found that units would pass the self-test but would fail in operation. Apparently there is a difference in timing between addressing a cell with a data call and addressing it with an execution cycle. To strengthen our testing program the self-test feature was altered so it now executes on each PROM. This self-test also checks the Revision number for each PROM to insure that the mix of PROMs in a terminal is a consistent set.

4.10.4 Memory Dump Facilities

Debugging firmware in a small machine is considerably more difficult than debugging SIGMA software. The tight limit on code space made it impossible to build in error detection and processing facilities such as appear in the SIGMA Error Package (see section 4.13.1, page 4-90). When the terminal recognizes an error condition it exits to a single error routine which prints on screen the address at which the code faulted and stops. This proved to be inadequate to diagnose the source of the error.

To solve this a terminal *Dump* facility was built. The firmware was changed to recognize a unique character string (two successive CANCEL characters) from the host computer. This sequence causes the terminal to send the contents of its entire RAM memory as a series of special Error notices, each with 32 bytes of data. A special utility, called DIAG, was provided for the TENEX operator, which he would use when a terminal failure occurred. It triggered this dump feature and stored the results for later System Programmer analysis.

4.10.5 Firmware Design

There are two processors in the HP 2649: the Intel 8080 and the Display processor (DP). The Intel 8080 is well documented in [21]. The Display processor drives the CRT. It contains the equivalent of a program counter (PC) which specifies the address of the next data fetch. At each fetch the PC is decremented and the this latter translates to "on what line on the screen it would be if the window were mapped." A separate dispatch is provided for each. This limited form of cursor control is the only way the host can establish what data is shown on screen. Since the user can scroll the screen contents, this is transient control at best. The host also has a dispatch to put the actual visible cursor into the desired mapped window (at the implicit cursor position).

4.10.2.5 Scrolling

The terminal's basic heuristic for mapped windows is to keep extra lines of text in margins above and below the lines that are on-screen. This lets the user scroll in either direction without having to go to the host for more data. Whenever the terminal assesses that a margin is getting too small, it will send a Vacancy notice to the host asking for more data for that margin. The terminal calculates the number of lines of data to ask for based on the size of the on screen area, the number of lines in the margin, and the amount of memory left in the terminal.

4.10.2.6 Memory management

The 12K bytes of display memory in the terminal are allocated as necessary for each dispatch. When the remaining memory is reduced to a prescribed limit, the terminal tries to reclaim memory from unmapped windows and, if that does not yield enough, from large margins of mapped windows. Memory is reclaimed by deleting domains and their contents from the edge of the margin and then telling the host through a Scroll notice. A Scroll notice identifies the last domain deleted and from which margin it came. It is important that the terminal be able to generate Scroll notices even when the keyboard is locked and the host is in control. It is during this state that the host will send the data that reaches the memory limit. The terminal must be free to reclaim memory right away in order to have room for the next dispatch, which may already be in the terminal's input buffer.

It is possible for the terminal to try to reclaim memory from the same margin the host is writing into. To prevent this the host can set special "No Reclaim" controls for each margin of each window. The terminal will not reclaim memory from a window margin so marked. The host must be careful not to leave these No Reclaim controls on, otherwise the terminal will quickly run out of margins from which it can reclaim memory, and the terminal memory will become full.

4.10.2.7 Communication discipline

Communication between the terminal and the host is really one computer talking to another. Each transmission must be error-free; otherwise the computer's model and the terminal's model may not match. To ensure the needed reliability of data across potentially noisy lines, a fully synchronized block retransmission protocol is used.

4.10.3 System Design

The firmware for the HP/MME terminal resides in Programmable Read Only Memory (PROM). This allows occasional changes to the code (requiring erasing the PROM and burning in new code) yet the terminal acts as though it has ROM memory. 32K bytes of memory are allocated for the firmware. 12K bytes are allocated for display list storage, and 8K bytes are for additional data (domains and window data and program variables).

data byte read is interpreted as 1) a symbol for display, 2) a display mode byte, 3) a special control byte, or 4) the first byte of a Jump operation. Table 4-1 illustrates the format of data interpretation by the Display processor.

Table 4-1: Display processor data interpretation

Bit	8	7	6	5	4	3	2	1	
	0	x	x	x	x	x	x	x	Symbol
	1	0	x	x	x	x	x	x	Display Mode
	1	1	0	0	x	x	x	x	Control
	1	1	0	1	x	x	x	x	Link
	1	1	1	0	x	x	x	x	Link
	1	1	1	1	x	x	x	x	Link

```
Symbol    =  128 ASCII characters

Display Mode
          Bit
             1 - Blink
             2 - Inverse Video
             3 - Underline
             4 - Half Bright
           5,6 - Character Set (select 1 of 4)

Control   =  Used for firmware.  No hardware operation except:
             Octal 314 = End of Line (EOL)
             Octal 316 = End of Page (EOP)
```

The Display processor is driven by control logic synchronized to the video display. Each refresh cycle (60 per second) starts the Display processor at the last byte in memory, which is the start of the display list (the character sequence to be presented). Normally, although not necessarily, this contains a Jump to some other location. Figure 4-13 illustrates the structure of the display list. This structure was chosen to be compatible with the structure used in HP's 2645 firmware. This turned out to be an unnecessary gesture towards compatibility.

The Display processor follows whatever the display list says. It puts the symbols it comes across into one of a pair of text line buffers. As one buffer is being loaded, the other is being unloaded--driving the video display. Display mode bytes set additional bits in the text line buffer that affect the appearance of the text when displayed. Upon reaching an EOL (end of line) special character or upon processing 80 symbols on the line the DP stops and awaits a text line synch pulse before starting the loading of the next text line. Note that this implies there can be blank space at the end of a line which does not take up memory.

When 24 text lines have been processed or if an EOP (end of page) byte is encountered before that, the DP stops and awaits the next frame synch signal to start again.

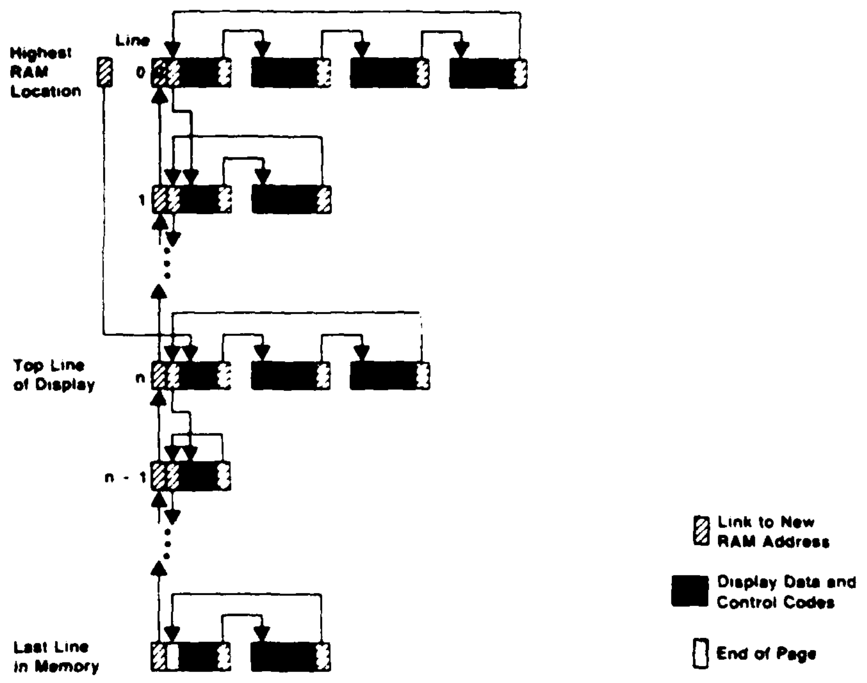


Figure 4-13: Display List structure

The contents of the screen are therefore entirely determined by the contents of the display list. This list is controlled by the ISI firmware. The address interpretation of the DP requires that the display list reside in the upper 12K bytes of the address space (52-64K).

4.10.6 Windows and Domains

Special variables are allocated for 8 windows. Window 0 is interpreted to be Flash Line data, so there are only 7 windows available to the application program. Each window is considered a contiguous linked display list that has potentially three parts: an upper margin, a visible portion, and a lower margin. The window always starts with a nondisplaying "dummy" first line and a nondisplaying dummy last line, created when the window is allocated and deleted when the window is deallocated. If the window is unmapped, there is no visible portion. The dummy first line is always in the upper margin and the dummy last line is always in either the visible or lower margin.

Window variables include such information as status (allocated or not, security level), window attributes, address of the domain block of the first domain in the window, and pointers into the display list for boundaries of the margins.

Domain information is stored in a separate area of RAM from the display list in 16-byte blocks (see Table 4-2). The domains for a given window are sequentially linked in the order they appear on screen. Domain blocks contain pointers to their beginning and end in the display list. The first domain always points to a dummy first domain which occupies the dummy first line. The first character in every domain is a display mode byte which determines the display mode for that domain. Any potential character position on screen that does not have . symbol defined for it in the display list is essentially undefined as far as the firmware (and

SIGMA) is concerned since all communication about characters is in terms of character position within a domain.

Table 4-2: Contents of domain block

Byte	
0,1	- Pointer to previous domain block
2,3	- Pointer to start of domain in display list
4,5	- Pointer to end of domain in display list
6,7	- ID of domain
8	- Format control
9	- Capabilities
10	- Row within margin of start of domain
11	- Margin for start of domain (0 = visible, 1 = upper, 2 = lower)
12,13	- Length of domain in bytes
14,15	- Pointer to next domain block

When the user edits the screen or the host adds, deletes, or alters domains, the firmware updates the display lists and domain lists accordingly, being careful to keep them entirely consistent.

4.10.7 Control Flow

The basic control flow of the terminal is fairly simple. Data input (from the host), data output (to the host), and keyboard scanning operate at interrupt level.

Data input interrupt processing entails putting the received character into the 256-byte circular input buffer, and interpreting the input byte in the context of the communication protocol. If the byte does not fit into the protocol properly, the protocol state is reset and a negative acknowledgment is stacked for the output process. If the byte is the last of a control sequence, the appropriate protocol state is set and the protocol response is stacked for output. If the byte is the last one in a successful dispatch, a flag (NEWDFL) is set for the dispatch reader and the protocol state is reset.

Data output and keyboard scanning are driven off a time interrupt that triggers every 10 milliseconds. The Data Comm Out process first checks to see if there are protocol control characters stacked for output. If so, it will take the top character off the stack, send it, and move the rest up. If no control characters are waiting, it will check for data characters to be output. These characters will be waiting in a 256-byte circular output buffer where they have been stacked by the HP/MME main program. The output process applies the communication protocol to the buffered data on the fly.

Keyboard scanning is done on each time interrupt (10 milliseconds). The state of the keyboard is compared with the state from the previous scan and an algorithm for two-key rollover is applied. The final interpretation of the keystroke is derived from a table. If the terminal state specifies that the keyboard is locked, the results of the scan are ignored (except for the ESC key, which is accepted at any time). The keystroke is then buffered temporarily in a one-byte buffer where it is picked up by the HP/MME main program (Read a Dispatch).

The main program consists of a simple loop. It starts by calling the Read a Dispatch (RAD) subroutine. RAD checks first for keyboard input. If there is a byte in the temporary buffer it is converted into the proper form of a dispatch and written into the Dispatch Buffer. If there is not keyboard input, the NEWDFL flag is tested to see if there is a new dispatch from the host waiting in the Input Buffer. If so, the dispatch is read into the Dispatch Buffer. If there is no Data Comm Input dispatch either, RAD halts and recycles after the next interrupt. Thus the subroutine RAD only returns to the main loop with a dispatch to be processed in the dispatch buffer. When it does return, the main loop interprets that dispatch through a vector table to the appropriate code for the particular dispatch. This dispatch process normally changes some state of the terminal (domain, display list, window variable, etc.). Sometimes it generates output for the host in the form of a notice. If the action called for by the dispatch cannot be performed, an Error notice is sent back to the host identifying the source of the error.

When the dispatch processing is complete, the main loop does some special processing to manage its memory. It first tests the amount of memory left in its display list free storage and its domain memory free storage. If either is below some prescribed minimum, each window is examined to see if there is data in an unmapped window or in the upper/lower margins of mapped windows that can be purged. Any data that is selected for purging is reported to the host in the form of a Scroll notice and the freed memory is returned to free storage. If the window attributes (which can constrain whether memory can be reclaimed from its margins) are such that the terminal cannot reclaim adequate memory, it will send the host a Terminal Full Error notice and subsequently reject any dispatch that tries to add or change domain contents.

After the test for reclaiming memory is complete, the main loop tests for vacancy conditions. A Vacancy notice is produced whenever the main loop finds a visible window (one which has some portion mapped on screen) with either an upper or lower margin which is less than a predetermined size. This condition is also limited by the vacancy control attributes assigned to each window.

After the vacancy test, the main loop returns to the top to test for the next dispatch. Thus the terminal is entirely driven by dispatches, and dispatches are generated from either the keyboard or Data Comm Input.

4.10.8 Dispatch Processing

Allocation, deallocation, mapping, and unmapping of windows require straightforward manipulation of the domain and window data structures. Cursor movement operations involve finding the appropriate enterable character in the display list. Scroll operations involve moving text lines to and from the visible region and the margins.

The most complex processing occurs when characters are added or deleted (domain dispatches from the host, insert and delete keyboard dispatches). These dispatches require altering the display list and domain structures to accommodate the altered data. Because of line wraparound, a change at the top of a window may propagate changes all the way to the end of the window. A single mechanism is employed to handle all text insertion and deletion. The following example of inserting a new, unformatted domain illustrates the use of this mechanism.

In the Create Domain dispatch the entire contents of the dispatch appear in the Dispatch Buffer. Ten bytes of domain data precede the actual text of the domain. This data is used to insert the new domain block into the domain list for the proper window. The text of the domain is then inserted into the display list. This is done by first finding the *Insertion point* in the display list (the end of the previous domain which may occur anywhere within a line of text). As illustrated in Figure 4-14 all text on the line after the Insertion point is copied to the end of the text of the new dispatch in the Dispatch Buffer. The Dispatch Buffer becomes a circular queue containing data to be inserted. It has a *Read Pointer* and a *Write Pointer*.

The contents of the queue from the Read Pointer are then written onto the text line starting at the Insertion point. As this text is written the word wraparound algorithm is dynamically applied. Typically only part of the queue contents will fit on the line. If this is the case, the next line is examined. If this line starts with a formatted domain (one which is defined to start at the left margin), then a new, empty blank line is inserted and the contents of the queue are written onto it and the process stops. If the next line is not the beginning of a formatted domain, then the contents of this line are copied to the end of the queue and data from the front of the queue are written in its place. Eventually this process will stop by encountering a formatted domain or the end of the window.

Text deletion is done in a similar manner, only in this case the Insertion point is at the beginning of the text to be deleted and an empty queue is loaded with the text on the line to the right of the last character to be deleted. See Figure 4-15.

4.11 SIGMA DATABASES

4.11.1 Message Database

Each SIGMA message is a single TENEX file. There are many more messages than can be held in a single TENEX directory. Therefore a series of TENEX directories are assigned for holding messages (SIGMA-MESSAGE1 through SIGMA-MESSAGESn). A message may migrate from one TENEX directory to another as it is updated. The message database then consists of the message files (one per message) and two special Message Directory files. Message Directory entries relate message identifiers (described below) to the location on the TENEX file system of the TENEX file containing the message. Additionally, they contain information concerning the archive status and security level of messages.

Message Identifiers

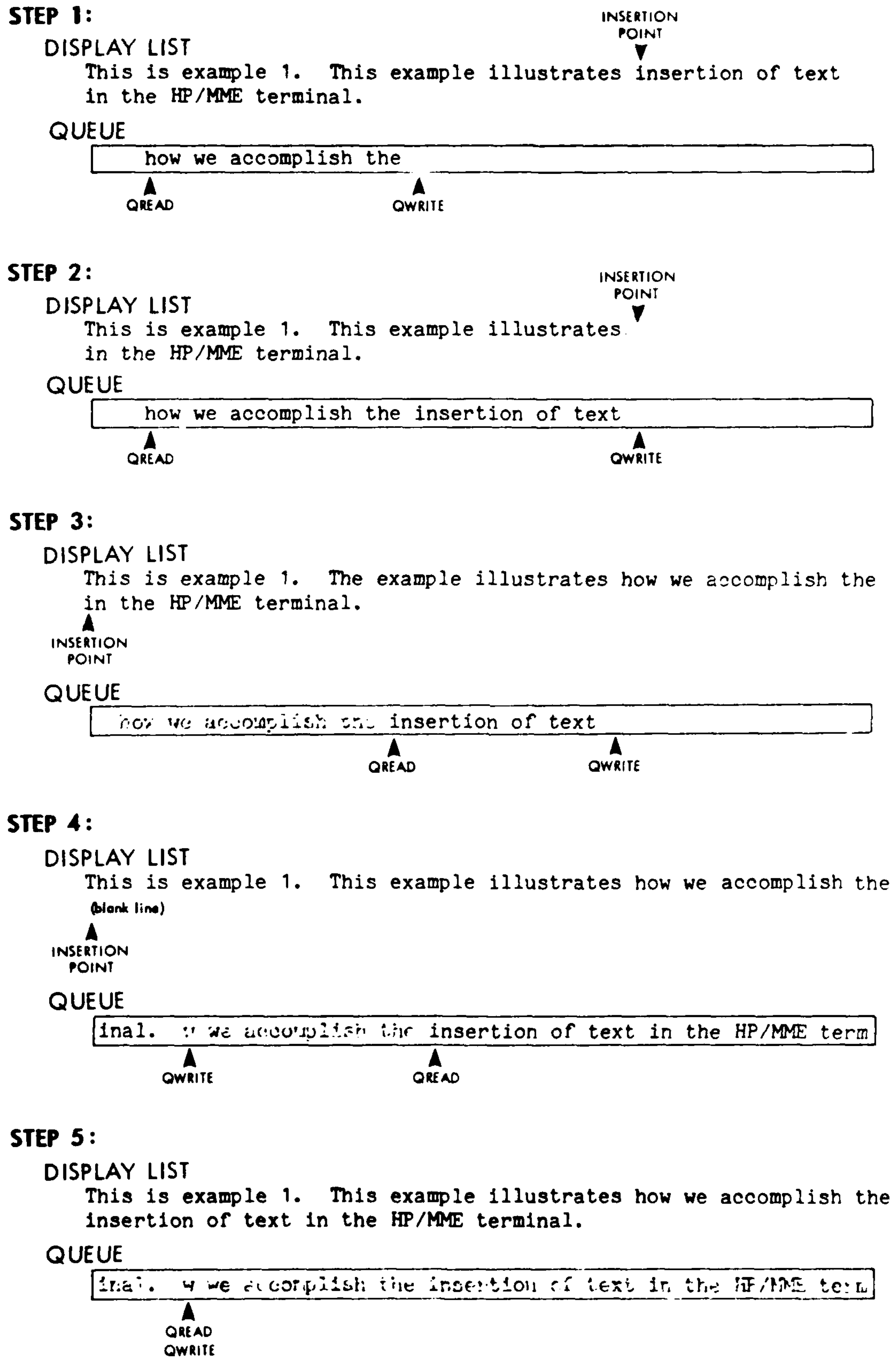
For all messages except received AUTODIN messages, a message identifier (hereafter referred to as "message ID") consists of the word pair:

(user identifier, date-time-group identifier).

The date-time-group identifier word is in TENEX'S internal date and time representation. The left half-word is the date (an integer count which started Nov 16, 1858, as day 1) and the right half-word is in seconds since midnight. If the internal message has been released (a Transmitted message) the least significant bit is set to a ONE, as a convenience testing. In-Preparation messages have ZERO LSB's. This distinction is in addition to the convention in SIGMA which ensures that In-Preparation messages have even-numbered date-time-groups and Transmitted internal messages have odd-numbered date-time-groups. Note that these date-time-groups are designated in terms of minutes, while the TENEX time representation is in seconds. For received AUTODIN messages, the ID consists of the word pair:

(distinguishing constant, assigned sequence number)

where the distinguishing constant is chosen so as not to be confused with any possible user identifier, and the reception process assigns a unique sequence number to any received AUTODIN message. Sequence numbers are used for received AUTODIN messages because date-time-groups are not necessarily unique for these messages since they are assigned by the originator and not by SIGMA.



STEP 1:

DISPLAY LIST

```
This is example 1.  This example illustrates insertion of text
in the HP/MME terminal.
```

QUEUE

```
    how we accomplish the
```

STEP 2:

DISPLAY LIST

```
This is example 1.  This example illustrates
in the HP/MME terminal.
```

QUEUE

```
    how we accomplish the insertion of text
```

STEP 3:

DISPLAY LIST

```
This is example 1.  The example illustrates how we accomplish the
in the HP/MME terminal.
```

QUEUE

```
  how we accomplish the insertion of text
```

STEP 4:

DISPLAY LIST

```
This is example 1.  This example illustrates how we accomplish the
```

QUEUE

```
inal.  w we accomplish the insertion of text in the HP/MME term
```

STEP 5:

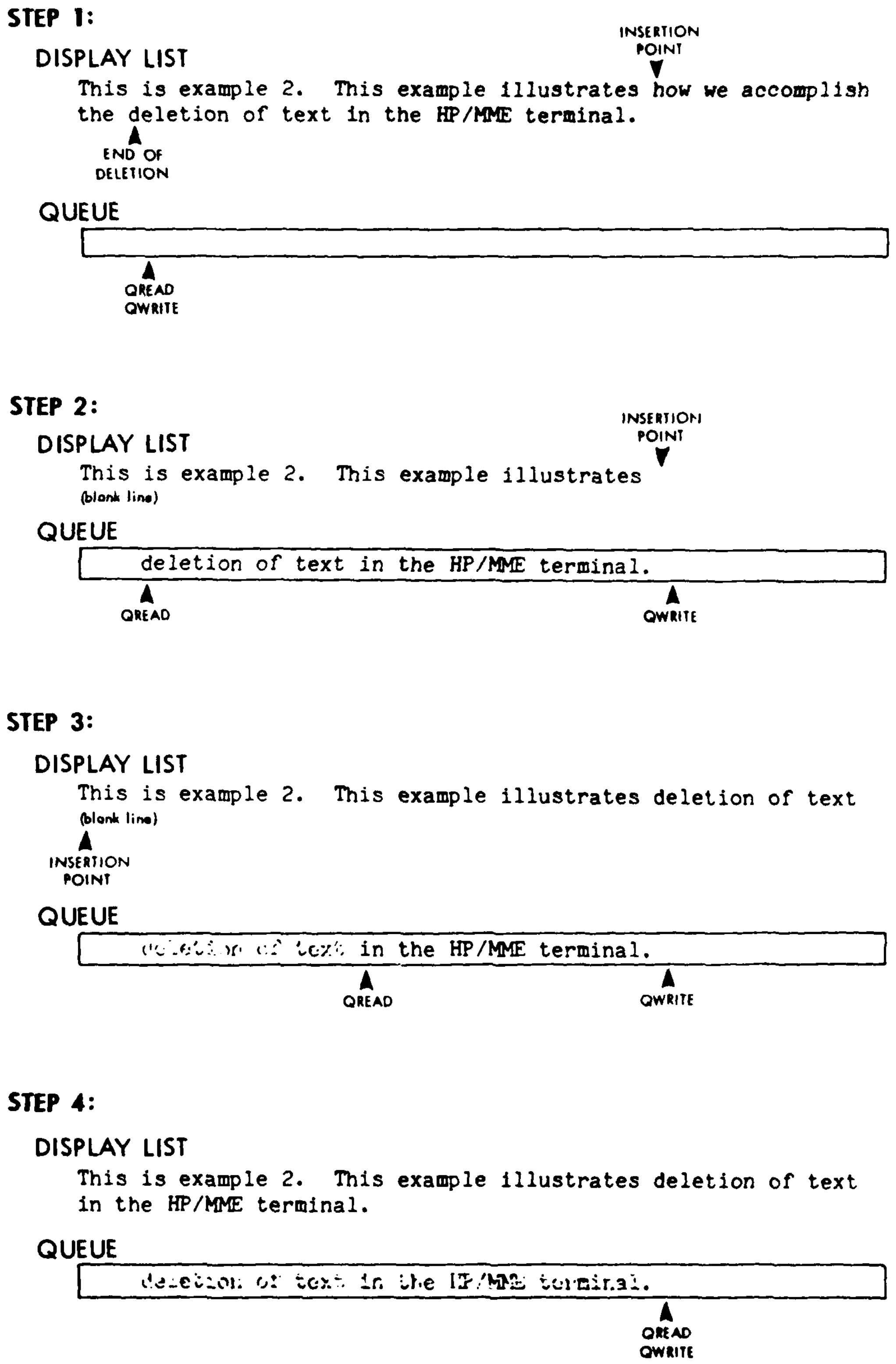
DISPLAY LIST

```
This is example 1.  This example illustrates how we accomplish the
insertion of text in the HP/MME terminal.
```

QUEUE

```
inal.  w we accomplish the insertion of text in the HP/MME term
```

Figure 4-14: Text insertion



STEP 1:

DISPLAY LIST

```
This is example 2.  This example illustrates how we accomplish
the deletion of text in the HP/MME terminal.
```

QUEUE

STEP 2:

DISPLAY LIST

```
This is example 2.  This example illustrates
(blank line)
```

QUEUE

```
deletion of text in the HP/MME terminal.
```

STEP 3:

DISPLAY LIST

```
This is example 2.  This example illustrates deletion of text
(blank line)
```

QUEUE

```
deletion of text in the HP/MME terminal.
```

STEP 4:

DISPLAY LIST

```
This is example 2.  This example illustrates deletion of text
in the HP/MME terminal.
```

QUEUE

```
deletion of text in the HP/MME terminal.
```

Figure 4-15: Text deletion

Message Directory Files

The directory files are

PSN.MD for received AUTODIN messages, and
DTG.MD for all other messages,

in directory <DAEMON-LOCAL-STATE>. They are maintained separately because the files are accessed in slightly different ways.

PSN.MD is accessed directly as a linear file without any explicit locks, in order to access the word entry at the offset in the file given by the sequence number.

DTG.MD is accessed by means of a hashing and search procedure, based on the date-time-group portion of the message ID, including locking mechanisms for access control. This is done because of the sparseness of the name space of actual date-time-groups in internal and preparation AUTODIN messages, so as to save space in the file.

Message Directory Entries

For all messages, a Message Directory entry is a single word, containing:

1. the TENEX directory number where the message may be found,

2. the TENEX directory number where the message is archived,

3. the security level of the message,

4. the archive status of the message (see section 4.12.8, page 4-85).

TENEX Location of Messages

Every message is a TENEX file. Each file is stored in one of a collection of directories; each file has a distinct name derived from the message ID as described below.

1. Received AUTODIN message files.

   The file name is

   PSNseqnr

   where seqnr is the decimal representation of the sequence number of the message. Thus, a received AUTODIN message with sequence number 2097855 is stored on file

   PSN2097855

   in the TENEX directory shown in the SIGMA Message Directory entry for the message.

2. Internal messages.

   The file name for internal message is of the form

   dirnum/364.ddhhmmZ/mon/yy

where:

- dirnnnn is the octal representation of the TENEX directory number of the author,
- ddhhmm is the initial six digits of the date-time group,
- mon is the three-letter representation for the month portion of the date-time group,
- yy is the two-digit representation for the year portion of the date-time group.

Example: if the message ID is

364,,1272
125034,,040070

then the file is

1272/364.090434Z/FEB/78

in the TENEX directory shown in the SIGMA Message Directory entry for the message.

If 1272 is the directory number for user JONES, a SIGMA user would access this message as

JONES 090434Z FEB 78;

since the time is even, this is an In-Preparation message.

4.11.2 Folder Database

Each SIGMA folder is a single TENEX file. There are more folders than can be held in a single TENEX directory. Therefore a series of TENEX directories are assigned for holding folders (UD-FOLDERS1 through UD-FOLDERSn). A folder may migrate from one TENEX directory to another as it is updated. The folder database then consists of the folder files, a Folder Directory file, and a Folder-Access file. The Folder Directory file contains the information needed to find a particular folder in the TENEX file system. The Folder-Access file provides control over the updating of the information in the Folder Directory file.

Folder ID

A Folder IS is a one-word quantity:

364,, folder number

Folder numbers are allocated as shown below (all numbers are in octal):

Number N:	Allocation:
0	not allocated
1-1777	Pending folder for user N
2000-7777	not allocated
10000-377777	User created folders
400000-777777	date files: in the form of the TENEX internal date representation for the day plus 400000.

Folder Directory

The Folder Directory is

<DAEMON-LOCAL-STATE>364.FOLDER-DIRECTORY

Folder Access File

The Folder Access file is

<DAEMON-LOCAL-STATE>364.ACCESS-LOCK

This file serves as a lock on the folder directory; it is opened for read/write if and only if an update is being done to a directory entry (e.g., by the Folder Daemon).

Folder Directory Entry

Word N of the Folder Directory contains information concerning folder N. If the folder is unassigned, the word contains 0. If the folder is reserved, the word contains -1. (Reserved folders are in the process of being created by some user-job or daemon process; the folder is reserved until the first update operation.)

For an assigned folder, the entry contains the following information:

1. the TENEX directory number of the owner of the folder. For datefiles, this quantity is 0.

2. the number of the TENEX directory containing the actual folder file.

3. a Boolean flag indicating whether the folder is busy (being updated); this is described in detail in [BUSYBIT].

TENEX Location of Folders

Every folder is a TENEX file. Each file is stored on one of a collection of directories, with the name

folder-number.364

where folder-number is the octal representation of the folder number.

For example, if the folder number is octal 222, then the folder is stored on TENEX file 222.364; if the folder directory entry is XWD (222,1725), then the folder is on the TENEX directory with directory number octal 1725, and the folder's owner is the user with TENEX directory number octal 222. This particular folder is the Pending folder for user 222.

4.12 THE DAEMONS

4.12.1 Introduction

The SIGMA message service consists of two major components: foreground user jobs, of which there is one per user, and a set of background daemon processes, which serve all the users. These daemons provide centralized control of the databases used in SIGMA and perform certain utility functions. The advantages of this background daemon processing are twofold. First, efficiency can be gained by deferring some processing to background, and second, sequential access to certain files can be guaranteed. The seven daemons which constitute this portion of the system are outlined below. (Detailed explanations can be found in later sections.)

1. MESSAGE daemon is responsible for the creation and updating of messages. It also transforms outgoing messages from internal SIGMA form into AUTODIN form as required. For internal messages (Memos and Notes) that are released, this daemon builds citations for updating the pending folders of the addresses. For messages that need to be coordinated, this daemon builds citations for the CITATION daemon for updating the Pending folders of the coordinating users

2. FOLDER daemon is responsible for creating, updating, and deleting folders.

3. CITATION daemon creates folder entries from citation requests put into its queue by user jobs and other daemons. *It accepts regular or fat citations and* updates the folder specified in the citation request.

4. RECEPTION daemon receives messages from LDMX/AUTODIN and transforms them into valid SIGMA messages. It stores the messages in the database and sends the appropriate citations.

5. ARCHIVE daemon issues retrieval requests to TENEX when it is necessary to retrieve archived messages. It issues appropriate citations when the messages are retrieved.

6. HARDCOPY daemon is a spooler for the hardcopy printing devices.

7. MAINTENANCE daemon is responsible for maintaining adequate file space by periodically removing deleted files.

Each daemon exists as a separate processor (i.e., TENEX job). The MESSAGE, FOLDER, ARCHIVE, and CITATION daemons are all driven by requests from the user jobs or other daemons. These requests are made in the form of "micros," interpretable blocks of data which define the request and contain all the necessary parameters, which the daemons receive through global queues. For example, any action by a user that requires modification of one of the databases will cause a set of micros to be written onto a queue for one of the daemons. The daemon will read the queue and perform the required functions. The RECEPTION daemon is driven by a file written by the LDMX, and the HARDCOPY daemon is driven by the appearance of a file (containing the text to be printed) in the hardcopy directory. The MAINTENANCE daemon is driven by a timer, i.e., it periodically removes deleted files.

4.12.2 Daemon Requirements

This section is based on the original daemon-requirements document. The points listed here preceded the implementation of the 2.0 version daemons and represented the target for their implementation. Some of the items were never implemented while others changed character during the experiment. These items are elaborated as they appear.

Daemon configuration

- Each daemon exists as a potentially separate process (i.e., Tenex job).

- Each daemon is able to restart itself in case of a system error.

- Requests to daemons from the user job come in the form of micros, an interpretable block of data which specifies a function and ALL its parameters (although intended to apply to Folder, Message, Archive, and Citation daemons. The Citation daemon only has one operation and so really is not micro-driven).

- All request-driven daemons receive and execute their micros through global queues.

- Interdaemon communication will be handled through the queue request procedure in the same way as user job requests.

Interface with the user job

- The user job communicates with the daemons through an explicit queue interface. The daemons therefore need not be up in order for the user job to run.

- The daemons do not need to know the names of the on-line SIGMA users, i.e., there is no log on/ log off to the daemons.

- The daemons keep no state information about the users. So, for example, in any daemon-to-user communication, all the necessary protocol information must be in the micro.

- No daemon request is lost until it is dealt with either by a successful DEQ or an error notification. Similarly, no user job request should fail during ENQ of a micro without notification to the user.

- SIGMA is not designed to run while the daemons are down, but if they are, the user job is able to operate gracefully.

- The user job has an easy way to tell if the daemons are up.

- Exception conditions to user requests are reported through pending files and existing alert procedures.

- The queue discipline for request-driven daemons should allow for some simple priority scheme. For example, certain real-time requests, those made by on-line SIGMA users, may be processed first. (This design requirement was never implemented because we could not find a user requirement that justified the complexities it introduced. It is still not clear whether it is justified.)

Daemon running environment

- The daemons are started and initialized by a Configuration Control Program which is run by the SIGMA operator.

- The state of each daemon is globally accessible. Each daemon has its own status page in a single status file. The associated page is identified by each daemon's unique index.

- No daemon depends on any other daemon being up (or even needs to check).

- Only one daemon may be modifying (i.e., updating) a folder or message at any given time.

- The daemons operate under a strict directory policy which prevents them from writing in read-only directories (e.g., those made up of BI I files), and in general dictates where working files can be written.

Micro-processing fork (MPFORK) environment

- Micros define the individual functions of an MPFORK. Each micro is self-contained, defining both its function and parameters. In order to process a request, the MPFORK uses a micro-decoder to "execute" each micro. A request has been successfully processed when each micro has been successfully decoded and executed.

- The entire micro-schema is flexible and general enough to support

  - easy addition, modification, and deletion of micros,

  - a table driven scheme which allows for easy test and debugging situations,

  - a description which is self documenting. (The file <IA-COMMON>MICDEF.SOURCE defines each daemon micro.)

- Each MPFORK uses a common micro decoder. The file <IA-COMMON>MI.DOC describes the MICRO package.

- Each daemon is required to do the right thing (i.e., maintain the proper state) in case of error. This might include DEQ or REQ or whatever in terms of an error during micro processing; but in any case, the daemons should avoid doing system errors when they can do smarter things.

Processing queues

- Queues should be robust:

  - No ENQ/DEQ should fail or be delayed indefinitely.

  - No micro-block is asked for more than twice.

  - The queue maintains its own boundaries around micros for resynchronizing reader/writer pointers when things go wrong.

  - Nothing can leave the queue without being reported.

  - The queue checks its own status at significant points, e.g., it checks if its pages get mapped away.

  - Overflow problems on ENQ should be handled smoothly. Among other things, this allows us to make the queues smaller, an important consideration for daemons that are tight for address space.

- Only the queue package can write in the queues. The queues are write protected (using TENEX facilities) during debugging mode.

- All queue state information is located in the queue.

- The queue package should support a simple priority scheme. (As mentioned above, this original requirement was not implemented.)

    - stopped or frozen individually or as a group.

  - General interface programs for

    - restarting trace files for particular daemons,

    - dumping daemon status,

    - dumping the state of the queues.

- There is a utility which archives messages. Its archival criterion is based on an operator-established parameter related to most recent use of a message. Archive retrieval is handled by the Archive Daemon. See section 4.12.8, page 4-85.

4.12.3 Daemon Environment

The daemons run under an environment which allows interactive monitoring and control. The operator may initialize, start, halt, and restart the daemons, either singly or as a group. Mechanisms are provided to allow a SIGMA operator to query the status of any or all of the daemons' operating components, including their current processing state, associated queues, etc.

Information regarding the status of the daemons and their interface to the SIGMA operator is embodied in the *Configuration Control Program* (CCP), while the actual control functions are implemented in the *Processor Controller* (PC) program.

4.12.3.1 Configuration Control Program

The CCP provides interactive monitoring and control of the SIGMA daemons. It serves two basic functions provided to the operator through a set of interactive commands:

1. Having access to the global databases used by the daemons, the CCP allows the operator to query the status of daemon processors, their queues, and associated state files.

2. By submitting requests to the PC, the CCP permits the operator to effect changes in the daemons' operating state.

These functions can be summarized as follows:

- Daemon activation: allow the operator to start, stop, resume, or kill any of the daemons.

- Status: query the operating status of the daemons and the current state of their request queues.

- State maintenance: initialize the daemons' running environment.

4.12.3.2 Processor Control

The PC is the common control and communication interface between the various SIGMA daemon processors, the Configuration Control Program (CCP), and the SIGMA operator. Running as the top fork of the daemon job, it is responsible for startup and initialization of the daemon, access to global data, communication with the SIGMA operator, and handling of unexpected errors.

Testing and performance

- The daemons needed a general error recovery and trace package more sophisticated than just ASSUME, SYSTEMERROR, and XTRACE (see section 4.13.1, page 4-90). This package conforms to the types of situations that the daemons are likely to encounter:

  - The daemons put all error notifications in a separate error log.

  - The daemons have a debug log in which to put general trace information.

  - The daemons have separate procedures for handling expected errors and unexpected ones (e.g., Illegal Instruction).

  - The daemon should be able to communicate directly with the SIGMA operator and halt, if necessary.

  - The daemons must have a test procedure which is independent of any user job. It is desirable for this procedure to be driven from a micro-catalogue for the daemons. (This requirement was really never met completely, although it was feasible. Daemons were generally tested by exercising them through a user job, simply because it was easier than writing a separate test program.)

  - Performance (i.e., PC samples) monitoring should be possible when testing in an off-line mode. (Again, although theoretically possible, no performance monitoring was done on the daemons.)

  - "Modules" were designed to accommodate their testing in isolation. This testability defines logical module separation.

  - Performance of the daemons should be wirebrushed to account for typical situations. For example, many file updates do not involve text changes, so this sort of update should be optimized to avoid copying the old text space. (This was never done. Our attention was directed to higher priority items.)

Utilities

- There should be a package of utilities for checking and dumping our databases, messages, files, user-state-files, etc. (The most important utilities were provided, but not a complete set, see section 4.13.6, page 4-104.)

- SIGMA operators have a sophisticated daemon interface program for the general problem of monitoring and controlling the daemons. The utilities provided include:

  - A Configuration Control Program (CCP) with which the daemons can be

    - initiated individually or as a group,

    - initialized at a level where timing dependent data can be created (or verified) easily to avoid synchronization problems,

- stopped or frozen individually or as a group.

- General interface programs for

  - restarting trace files for particular daemons,

  - dumping daemon status,

  - dumping the state of the queues.

• There is a utility which archives messages. Its archival criterion is based on an operator-established parameter related to most recent use of a message. Archive retrieval is handled by the Archive Daemon. See section 4.12.8, page 4-85.

4.12.3 Daemon Environment

The daemons run under an environment which allows interactive monitoring and control. The operator may initialize, start, halt, and restart the daemons, either singly or as a group. Mechanisms are provided to allow a SIGMA operator to query the status of any or all of the daemons' operating components, including their current processing state, associated queues, etc.

Information regarding the status of the daemons and their interface to the SIGMA operator is embodied in the *Configuration Control Program* (CCP), while the actual control functions are implemented in the *Processor Controller* (PC) program.

4.12.3.1 Configuration Control Program

The CCP provides interactive monitoring and control of the SIGMA daemons. It serves two basic functions provided to the operator through a set of interactive commands:

1. Having access to the global databases used by the daemons, the CCP allows the operator to query the status of daemon processors, their queues, and associated state files.

2. By submitting requests to the PC, the CCP permits the operator to effect changes in the daemons' operating state.

These functions can be summarized as follows:

- Daemon activation: allow the operator to start, stop, resume, or kill any of the daemons.

- Status: query the operating status of the daemons and the current state of their request queues.

- State maintenance: initialize the daemons' running environment.

4.12.3.2 Processor Control

The PC is the common control and communication interface between the various SIGMA daemon processors, the Configuration Control Program (CCP), and the SIGMA operator. Running as the top fork of the daemon job, it is responsible for startup and initialization of the daemon, access to global data, communication with the SIGMA operator, and handling of unexpected errors.

Only one PC program is substantiated for each of the daemons to be run. Because it must be able to control all of the daemons, the PC has a very general model of their tasks and requires an individual set of interface specifications for each daemon. In addition, the PC sets up a general running environment for the daemons which allows them to both access global data of interest and communicate with the outside world.

Physically, the PC is broken into two parts: a self-contained program, PC.SAV, which runs as the top fork of the daemon's job, and PCPINT.REL, which is the main program for the lower fork for each specific daemon job and serves as the interface between the daemon-specific code and the rest of the PC. These are described in more detail below.

4.12.3.2.1 Overall Processor Control--PC.SAV

This program is started whenever a daemon is to be run. It determines whether it is running in production mode (detached) or checkout mode (attached to a terminal). When running in production mode, it is passed the *PINDEX* (daemon index) of the daemon to be started as part of its startup parameters. When running in checkout mode, the PC interacts with the user to determine which daemon is to be checked out, and communicates various kinds of status information to the user as the daemon runs.

When started, the PC determines which daemon is to be run (as described above), ensures that there is not another instance of that daemon already running (and halts if so), loads and initializes the daemon, and starts it. While the daemon is running, the PC has two main functions:

- The PC constantly monitors the daemon's state to determine whether an error has occurred. If so, the PC stops the daemon (if it has not already stopped), logs the error, and notifies the operator.

- The PC watches for commands issued by the operator through the CCP. These commands can ask the PC to stop the daemon, log out the daemon job, and other maintenance requests. When possible, the PC attempts to allow the daemon to finish its in-progress task before any interruption.

4.12.3.2.2 The Daemon Interface--PCPINT.REL

This module is loaded with the functional daemon code, and serves as both the main program for the daemon and its interface to the PC and the outside world. When the daemon is started by the PC, this module takes control and performs necessary initialization, connects to global daemon state information (making it accessible to the daemon code), and starts the daemon execution. During that execution, should any fatal error occur the PC gains control through the Error Package (to allow it to clean up whatever state might still be left "dirty" by the daemon) and notifies the top PC fork of the error condition.

In order for PCPINT to be implemented in a daemon-independent way, the following routines must be defined in the daemon code:

?P.INIT	This routine is called by PCPINT during daemon initialization, just after the PC-specific initialization has been performed. ?P.INIT should perform any daemon-specific initialization.
?P.PROCESS	This routine is called after initialization is complete, and performs the actual processing. It is never expected to return.
?P.CLEANUP	This routine is called if a fatal error is detected in the daemon. It is expected to know both what state the daemon must clean up and how to do it.

In addition to control, PCPINT provides other services to its daemon: it allows access to various state and running information, and it provides several utility services (through supplied subroutine calls) which allow the daemon to define clean interruption points, broadcast its operating state, and send messages to the SIGMA operator.

4.12.4 Message Daemon

The Message daemon is responsible for maintaining the SIGMA message database. It is a micro-driven processor. That is, requests for service are in the form of blocks of information (micros) which specify the operation to be performed plus any necessary parameters. Requests are enqueued in the Message daemon queue by the requesting process. Each queue entry is independent of all others in terms of queue organization (e.g., there is no priority scheme).

Since the Message daemon spends most of its time accessing messages, it loads its own copy of MSGMOD (the same message access module used by the user job) as a lower fork.

4.12.4.1 Functional description

The Message daemon performs the following functions:

- It records messages in the SIGMA message database (RECORD-MESSAGE).
- It releases messages (RELEASE-MESSAGE).
- It updates messages with delta file information (DELTA-UPDATE).
- It starts the coordination process by building *For_ Chop* citations for each member of the coordination list (COORDINATE-MESSAGE).
- It builds citations for files into which messages are to be entered (CITE-TO-FILE).
- It builds *Chopped* citations for delegators indicating that a message has been reviewed and either approved or disapproved, specifying the chopping user (CHOP-MESSAGE).
- It adds users to address fields of a message and builds citations for each new addressee (APPEND-ADDRESSEES).

The citations built by the Message daemon are all enqueued for processing by the Citation daemon (see section 4.12.6, page 4-81).

4.12.4.2 Daemon operation

The main control loop for the Message daemon is very simple. It looks to see if there are any micros in the Message queue. If so, routine ?MEDECODE, is called to decode them and call the appropriate action routines. If no error occurs, the micro is dequeued and the Message daemon proceeds to the next micro. When the queue is found to be empty, the Message daemon waits 20 seconds before looking again.

The error handling is sensitive to two issues: the possibility of the message busy bit being on (section 4.8.2, page 4-38), and the proper maintenance of the internal daemon state. For example, in addition to error logging if the micro in which the error occurs is of the update variety, the following is done:

- The associated delta file is deleted. It is of no value, since the update failed.

- The Message daemon's internal MSGMOD state is cleared.

- The message in question is read in so the busy bit for the specified user may be turned off.

- The Message daemon's internal MSGMOD state is cleared again.

- The micro is dequeued.

These last two steps are normal procedure for all micros, whether in error or not.

This busy-bit procedure is also called when the queue package is going to automatically dequeue a micro because of repeated failures (section 4.13.5.5.1, page 4-104). In both cases (error or auto-dequeue), if we fail to turn off the busy bit, the user who initiated the update request will be forever locked out from the message, a situation which will require operator intervention to remedy.

The daemon state cleanup comprises:

- unlocking the message directories,

- clearing the internal MSGMOD state,

- unlocking the queue.

These procedures evolved from CINCPAC experience and emphasize the need for robust background processors.

The specific functions of the Message daemon are performed as follows:

RECORD-MESSAGE

The message is opened, its status is set to "not transmitted," and its precedence, type, and security are properly set. The message is then recorded in the message database as existing in the specified directory with the specified security. Finally, the message is closed.

RELEASE-MESSAGE

The preparation version of the message is read in and checked to verify that it has not already been released. Then a released version of the message is prepared with a "transmitted" status.

Incoming citations are built for all users specified on the TO and INFO lists (MEMOS and NOTES). *Back_Copy* citations are built for the releaser (MEMOS and NOTES) plus all users specified on the releaser's coordination list (MEMOS). If the message is of type FORMAL (MEMO), then J301 is added to the backcopy list. The released message is added to the SIGMA message database. Finally, the preparation version of the message is marked as released.

For AUTODIN messages the Message daemon does the above operations with the following differences. The Message daemon does not store the transmitted message into the SIGMA message database. Instead it creates a sequential text file in a format acceptable to the LDMX server, for transmission by that server to the LDMX. The Message daemon does not generate *Back_Copy* citations for the transmitted message. *Back_copy* citations are generated by the Reception daemon on receipt from the LDMX server of the backcopy AUTODIN message from the LDMX.

DELTA-UPDATE
The message to be updated is read in. If it is a Preparation Message which has been transmitted, an *Xmit_Fail* citation is issued and processing is aborted. Otherwise the information in the delta file is merged with the message. The message is updated into a new version of the same message. *File_Copy* citations are built for any users which have been specified to receive file copies of the message. Finally, the message is cleared to a not-busy state and the delta file is deleted.

COORDINATE-MESSAGE
The Message daemon reads in the message being coordinated. It then builds *For_Chop* or *For_ Release* citations for all users specified in the chop and release message fields, respectively. Finally, the message is cleared.

CITE-TO-FILE Citations of the type specified (*File_Copy*, *For_Action*, or *For_Info*) are built for each file given in a list of file IDs.

CHOP-MESSAGE
If the message specified is a Preparation message which has already been transmitted, an *Xmit_Fail* citation is issued and processing is aborted. Otherwise the information in the delta file is merged with the message. Then a *Chopped* citation is sent to the delegator. The message is updated into a new version of the same message. Finally, the message is cleared to a not-busy state and the delta file is deleted.

APPEND-ADDRESSEES
The specified message is read in. The specified users are appended to the specified fields of the message. If no message field is specified, then a check is made for the presence of ACTION, COG, and ORIG fields in the message, and the users are appended to the first one found to exist. If none of these fields exists in the message, then an ACTION field is created and the users are added to it. The message is updated into a new version of the same message. Citations of the appropriate type (i.e., *For_Info* or *For_Action*) are built for each of the specified users. Finally, the message is cleared to a not-busy state.

4.12.5 Folder Daemon

The Folder daemon is responsible for maintaining the SIGMA folder database. Like the Message daemon, it is a micro-driven processor, to which other processes enqueue requests. Since it spends most of its time manipulating folders, it loads its own copy of FACMOD (the folder access module used by the user job) as a lower fork.

4.12.5.1 Functional description

The Folder daemon performs the following functions:

- It records folders in the SIGMA folder database (RECORD-FOLDER).
- It deletes folders from the SIGMA folder database (DELETE-FOLDER).
- It updates folders with delta file information (DELTA-UPDATE).
- It creates inboxes (pending folders) for specified users (CREATE-INBOX).

4.12.5.2 Daemon operation

The Folder daemon is a micro-driven processor. Requests for services are put in the Folder daemon queue; each entry is independent of all others in terms of queue organization. The main control loop for Folder daemon is quite simple. It looks to see if there are any micros in the Folder queue. If so, routine ?FIXDECODE, is called to decode them and call the appropriate action routines. If no error occurs, the micro is dequeued and the Folder daemon proceeds to the next micro. When the queue is found to be empty, the Folder daemon waits 20 seconds before looking again.

As with messages the error handling is sensitive to busy bits and internal state. If a micro fails, the following is done:

- If the busy bit is on (indicated by one of the micro parameters), it is turned off.
- The locking request mechanism with the Citation daemon is cleared.
- The folder directory is unlocked.
- The queue is unlocked.
- If the folder file is open, it is closed.

In the case of automatic dequeuing of the micro by the queue package (section 4.13.5.5.1, page 4-104), the micro's busy bit parameter is checked and if on, the busy bit is turned off. Note that the busy bit for folders is located in the folder directory (for the owner only), while messages have busy bits for potentially every user.

The specific functions of the Folder daemon are performed as follows.

RECORD-FOLDER
The name of the TENEX file corresponding to the specified folder ID is generated. The file directory is checked to verify that that folder name has not already been used. If not, a new file in a folder directory is created with the specified name. Then the image file is renamed to the real file name and is entered into the folder directory.

DELETE-FOLDER
The Folder daemon verifies that the user requesting the delete is the owner of the folder. It requests that the Citation daemon give up the folder if it is currently processing it. It then finds the real TENEX file name and directory of the folder and connects to that

directory. Finally, it gets rid of the entry in the SIGMA folder directory and deletes the TENEX file.

DELTA-UPDATE
The Folder daemon first requests that the Citation daemon give up the folder if it is currently processing it. It then finds, opens, and checks the consistency of the delta file. It brings up the folder and merges the delta information into it. The folder is then updated to a new version, and the delta file is deleted.

CREATE-INBOX The Folder daemon verifies that the specified user ID refers to a legal user and that the user does not already have a pending folder. It then creates the image of a pending folder and performs an update to put the image into a real file.

4.12.6 Citation Daemon

4.12.6.1 Functional description

The queue-driven Citation daemon's primary task is to create folder entries from citations and put them into designated folders. Each queue element (citation) contains the destination file as well as the data required to build the entry. In performing this function the Citation daemon deals with certain special performance issues:

- It optimizes the handling of citations destined for a specific folder.
- When processing a SIGMA folder, it is sensitive to a Folder daemon request for the same folder; that is, the Citation daemon will give up a folder if another daemon needs it.
- Its folder update procedure is identical to that of the Folder daemon.

4.12.6.2 Citation structure

In order to describe the operation of the Citation daemon it is necessary to understand something of the structure of a citation. There are two basic kinds of citations, referred to as small and fat. Small citations give only some of the information necessary to build the folder entry. This includes citation type, message ID, and destination folder. The actual strings required for the folder entry are built by opening the message referred to in the small citation and extracting them from it. Fat citations include all the information required to build the folder entry including the strings. The reason for the two kinds of citations is strictly efficiency. Often the process enqueuing a citation will already have the message open. Thus, it can directly provide all the information the Citation daemon needs to build the folder entry, eliminating the need to open the message again.

4.12.6.3 Daemon operation

The Citation daemon is a queue-driven processor. Periodically it looks to see if the citation queue, <DAEMON-LOCAL-STATE>QUEUE.CITATION, is nonempty. It does this by asking the queue package to return the (physically) first entry in the queue (if the folder named in the citation is being updated by the Folder daemon, then the "next" citation with a different folder ID is asked for). It then opens the Pending folder named in the citation. The daemon will then try to process all the citations (one at a time) currently in the queue for this folder until either:

- the folder daemon requests this particular folder,

- a prespecified maximum number of citations processed during one update is reached,

- all the citations destined for this folder are processed.

At this point a folder update is done, completing the cycle for this pending folder. When the citation queue is found to be empty, the daemon will wait 10 seconds before checking again.

The Citation daemon queue is organized somewhat differently from the other daemon queues. This is due to the requirement that it optimize handling of citations destined for the same folder. Instead of having separate, independent entries in the queue, all entries going to the same folder are linked together. A special queue package routine returns the next entry on the current list instead of the next entry in physical order.

The update process in the Citation daemon high-level control structure performs several functions. First it does the actual FACMOD folder update. If the update succeeds, it then closes the subject folder and begins alert processing if the destination user is on-line. If he is, the associated user's alert queue is mapped so that during the dequeue cycle, the citation blocks can be enqueued into the alert queue. Then all entries that were processed for the just-updated folder are dequeued from the citation queue (and enqueued into the user's alert queue if the user is on-line). Note that this implies that if any problems had occurred up to and through the update, the citations would have been preserved in the queue for appropriate retry, error handling, etc. Lastly, the user's alert queue is unmapped and closed.

The routine that puts citations into folders performs several duties. It first converts a small citation into a fat one if necessary. It then builds a folder entry from the citation and appends it to the destination folder. It also leaves the fat version of the citation in a buffer that the main Citation daemon knows about, and this is what is later enqueued into the user's alert queue.

Because of efficiency considerations and the small size of the Citation daemon, MSGMOD is directly loaded. FACMOD, however, resides in a lower fork. (For a discussion of MSGMOD and FACMOD see section 4.8, page 4-37.)

4.12.7 Reception Daemon

4.12.7.1 Functional description

The Reception daemon performs the following functions:

- It parses AUTODIN messages from the LDMX and puts them into SIGMA format. It reads these AUTODIN messages from a sequential text file produced by the LDMX server and creates a SIGMA message file for each message, entering it into the SIGMA message database. If the Reception daemon cannot parse the message from the LDMX, or if it determines that the message should not be made available through SIGMA (because of restricted access), it appends the sequential text for the message to one of a set of special files.

- It builds citations for SIGMA users and/or datefiles, based on routing specified in the message and by a distribution control file. The control file may specify particular routing for various classes of messages based on any one of several criteria: security level, precedence, or particular text strings' presence in the message body. The particular routing may include any of the following: forcing particular user(s) to be included, excluding particular user(s), including or excluding the distribution specified in the message, rerouting the message distribution, including or excluding datefiles, or rejecting the message outright (the message is to be appended to a special "turn-around" file).

- It brings any AUTODIN service messages in the file prepared by the LDMX server to the attention of the TENEX operator for action.

4.12.7.2 Daemon operation

The Reception daemon is driven by a file written by the LDMX server. Periodically it looks to see if this file, <LDMX-OPERATOR>LDMX.OUT, is nonempty. If it is, it opens the file for read, parses the sequential text file for one message, builds a SIGMA message from the AUTODIN message it has read, closes the file, and enters the resulting SIGMA message into the message database. It then builds and enqueues, for the Citation daemon, citations for all of the SIGMA users that are named in the distribution lists of the message, according to the routing instructions contained in the file ACCESS.CONTROL, or in the file AUTODIN-RECEIVED-ROUTING.LEXICON. It then waits for a short interval (five seconds), and repeats the above procedure for the next message in the file, if any. During this interval, the LDMX server may append new messages to the file.

The Reception daemon may be unable to open the file when it determines that the file is not empty, most likely because the LDMX server is still appending new messages to the file and has not yet closed the file. In this case, the Reception daemon keeps trying at five-second intervals to open the file, until it succeeds; it then proceeds to process the file as outlined above.

If the LDMX.OUT file is empty, the Reception daemon waits for two minutes and then looks again.

The Reception daemon notes, when opening the file, where the next message is to be found by computing from the length of the message (contained as part of a line of text at the beginning of every message) and the current position of this message. It records this information in the user-settable word of the file descriptor block (FDB) for LDMX.OUT. When the process later opens the LDMX.OUT file for read, it positions its pointer in the file according to the value stored in this user-settable word of the file descriptor block.

When the Reception process finishes processing the last message contained in the LDMX.OUT file, it empties the file of all its pages (using the PMAP JSYS), resets the end-of-file pointer to 0, resets the user-settable word to 0, and closes the file. It then waits for two minutes and looks again for messages.

If the Reception process finds a problem with the format of some part of the message in LDMX.OUT, it does not produce a SIGMA message but instead appends the message to the file LDMX.TURNAROUND. Additionally, the daemon may find a basic problem with the file, in which the initial part (known as the "RFC680 header line") of what should be a message does not have the proper information concerning the date and time of arrival, message length, or message precedence. When this occurs there is serious doubt about the integrity of the file, at least that point in the file at which this problem occurs. Consequently, the daemon copies the remainder of the file into LDMX.TURNAROUND, first appending a correct header line at the beginning of the incorrect part of the file. Finally, the daemon notifies the operator, through the CCP, of all messages turned around.

The process logs all messages processed, whether as SIGMA messages or as turned-around messages, in the file LOG.RECEIVED-AUTODIN;yyddd, where yy is the year in two digits, ddd is the Julian date.

For controlling the distribution of citations, the files ACCESS.CONTROL and AUTODIN-RECEIVED-ROUTING.LEXICON are used. ACCESS.CONTROL is created by a utility program, DACCESS.SAV from a standard text file which specifies, using a simple grammar, the kinds of criteria for which a message will receive controlled distribution. For each criterion the controlled distribution is specified.

As each message is read from the sequential text file, it is tested against each of the criteria in order, until either a match is found or no more criteria remain to be tested. If a matching criterion is found, the distribution specification for that criterion is used for controlling distribution of citations. If the distribution specified is to reject the message to the turn-around file, the message is appended to the file LDMX.RESTRICTED-ACCESS, and no SIGMA message file is built nor are citations sent; otherwise, the distribution of citations is carried out as specified.

If no match is found, a second file, AUTODIN-RECEIVED-ROUTING.LEXICON, is used to determine, from the message-specified distribution, the SIGMA distribution, as follows (additionally, datefile distribution will be made):

> For each office code specified for distribution in the message, if it is found as a target in the lexicon, the definitions for the target are the distribution; if not, if that code is nevertheless also a SIGMA user name, that user will receive a citation; otherwise, no distribution on that code will be made.

Distribution to datefiles is made on the basis of the date-time-groups (DTGs) of the AUTODIN message identifications for the message. If the message has not been readdressed (i.e., forwarded to this message center from another message center on the AUTODIN network), there wil' be only one DTG, that assigned by the message originator. If it has been readdressed, there will also be a DTG for each AUTODIN message handler that readdressed the message. Each DTG for the message will result in a citation's being sent to the datefile corresponding to the date portion of the DTG.

Occasionally, SIGMA receives a service message, which is usually a message from the LDMX operator to the SIGMA system concerning a matter requiring SIGMA operator action. For example, such a message might concern a problem with an outgoing message originated by a SIGMA user (perhaps an error in the message's format). Such messages must usually be handled manually by the SIGMA operator, since they relate to message communication issues and would not be of help to most SIGMA users.

The Reception daemon recognizes a service message by the presence of a special four-letter code in the first line of the message. On recognition, the daemon handles the service message as follows:

- it prints the enti:e message on the TENEX line printer, appending a heading line showing the time and date of receipt;

- it appends the entire message to LDMX.SERVICE-MESSAGE for possible future reference; and

- it notifies the operator through the CCP interface of the receipt of the service message.

No SIGMA message is built from a service message, nor are any citations sent concerning the message.

A message may be marked for "early archiving," i.e., for archiving at a shorter than regular interval, by setting the user-settable word of the message file's descriptor block to a nonzero value. The value so set is the number of days after the last access after which the message should be archived. The only process which so marks messages is the Reception process. On receipt of a complete AUTODIN message, the Reception process matches the message originator against a lexicon of message originators and corresponding archive intervals. If a match is found, the process writes the corresponding interval into the user-settable word.

4.12.7.3 Efficiency considerations

The reception process is expected to handle the major portion of the AUTODIN message traffic; hence, the process needs to be efficient. For this reason, the MSGMOD code is not run as an inferior fork, but instead is directly loaded into the reception process itself. The saving in processor time from eliminating scheduler overhead has turned out to be significant. For the same reason the input file, <LDMX-OPERATOR>LDMX.OUT, is read not using the sequential input monitor calls of TENEX, SIN and BIN, but rather by mapping the pages of the file directly.

4.12.8 Archive Daemon

4.12.8.1 Functional description

The Archive daemon's main function is to retrieve messages which have been archived. Specifically it does the following:

- It makes requests of the TENEX system for the retrieval of message (and, potentially, folder) files from the archive as requested by the SIGMA users. (The actual retrieval of message files from the archive is made by the TENEX program BSYS. The Archive daemon just makes requests of BSYS.)

- It notifies the requesting user of success or failure of the retrieval by sending a citation to that user's Pending file.

4.12.8.2 The archive process

Before describing the operation of the Archive daemon, it will be helpful to explain the process by which messages are archived. A program, ASCAN.SAV, is run from time to time by the TENEX operator. It marks SIGMA message (and potentially folder) files to be archived.

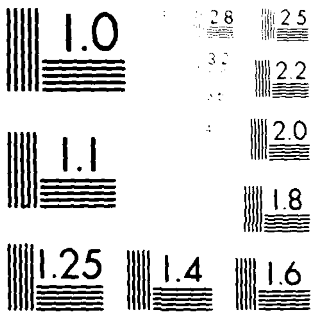
ASCAN determines from the operator the interval in days for which messages are to be retained on-line (i.e., not archived). It then makes a pass through all the on-line messages. For each message it determines whether to mark the file for archiving based on the date of last access (this is the more recent of the date of read and the date of write). The message is archived if the number of days since its last access exceeds the archive interval. The message may also be marked for early archiving (see 4.12.7.2), by having the user-settable word in its file descriptor block set to nonzero. The value is the intended archive interval for the message (for example, if the user-settable word is 3, the interval is three days, so that the message should be archived in three days from the date of last access). If this user-settable word is nonzero, the archive interval is taken to be the minimum of it and the operator specified interval.

When ASCAN finds a message to archive, it opens the message database that has the entry for the message. (The message database will be temporarily inaccessible while it is open; however, any process (including this one, if another process has the database open) will wait until the database is again free.) Each message in the message database (section 4.11.1, page 4-65) contains the following information:

AD-A116 360   UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC   F/G 17/2
SIGMA FINAL REPORT: VOLUME V, PART 4. THE DESIGN OF SIGMA.(U)
MAY 82   R STOTZ, D WILCZYNSKI, S FINKEL   DAHC15-72-C-0308
UNCLASSIFIED   ISI/RR-82-95-VOL-5-PT-4   NL

2 OF 2
AD A
116360

END
DATE
FILMED
08-82
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

- The message-directory number for the message;

- The archive-directory number for the message (if appropriate);

- The state of the message, which can be any one of the following:

  1. The message is in the message-directory, on-line.

  2. The message is in the archive-directory, possibly archived.

  3. The message, archived, has been requested for retrieval.

  4. The message was not retrievable from the archive (this state indicates that the message has somehow been lost, since it should be in the archive, but is not).

- The security level of the message.

ASCAN will then attempt to rename the message file to one of a number of special directories, known here as IA-Archive directories. Their purpose is to act as directories from which files are archived and to which files are retrieved. No messages are ever archived from or retrieved to the original message (IA-Message) directories directly; retrieval and archival always proceed through an IA-Archive directory. Messages are always accessed in the service from a regular message directory, never from an IA-Archive directory. Consequently, files are moved (by renaming) to or from an IA-Archive directory in the process of archival or retrieval.

Once the message file has been successfully moved to an IA-Archive directory the message database is appropriately changed to reflect this fact. The message database is then released.

4.12.8.3 Daemon operation

The Archive daemon is a micro-driven processor. When it finds a retrieval request micro in its queue, it checks whether the message specified is already on disk, either in an IA-Message directory or an IA-Archive directory. If it is, the daemon moves the file, if necessary, to the IA-Message directory, updates the message database, and sends a *Retrieved* citation to the user. If the message is not on disk, but is shown in the database as being archived and not already requested from the archive, it makes a TENEX BSYS request to retrieve the message and changes the message state to show that it has been requested for retrieval.

When the file is retrieved the daemon moves the file from the IA-Archive directory to the IA-Message directory, updates the database, and enqueues a *Retrieved* citation for the requesting user.

Since an indefinite interval of time may elapse between the issuing of a BSYS retrieval request and the completion of the retrieval operation, the daemon enqueues on its own queue a check request micro, to cause it to check later for the retrieval of the TENEX file from the archive (the daemon deletes the user's original retrieval request micro). Thus the daemon's queue is used as a buffer for two kinds of micros: the users' retrieval requests and the daemon's check requests.

When processing a check micro, the daemon acts in much the same way as for a retrieval micro, except that only the presence of the file on disk is checked, and there is no BSYS retrieval request (since a request has already been made).

When the daemon processes a request micro (either retrieval or check) and determines that it is as yet unsatisfied, it requeues the request as a check request micro. An end-of-queue-mark micro is written to denote the end of the current list of micros to process on any given processing cycle. The daemon waits for 75 seconds from the end of one processing cycle before beginning a new cycle.

In the event that the request to retrieve the message file fails (this might happen if the TENEX archive database file is damaged), the daemon changes the message state in the database to show the message as nonretrievable and enqueues a retrieval-failure citation for the requesting user (this is a particular kind of error citation), detailing the circumstances of the failure.

4.12.8.4 BSYS retrieval request generation

Since no modification to the BSYS system was allowed, the actual request is made by the program LOOKUP, the standard program for requesting retrievals from the TENEX archive. This program is started at a special entry point, in an inferior process, once for each retrieval request. The Archive daemon process and this process share one page to communicate the name of the file to be retrieved and the error string returned by LOOKUP if the request cannot be satisfied (e.g., this string is generated if the file to be retrieved is not found in the TENEX archive). If the retrieval is successful, the LOOKUP program returns success. If LOOKUP is not successful, an error string or an operating system error code is returned. In this case, the Archive daemon enqueues an error citation to indicate to the requesting user that the retrieval cannot be made; additionally, the daemon changes the state of the message in the database to show that it is not retrievable. The daemon logs the error for later analysis by system personnel.

4.12.9 Hardcopy Daemon

4.12.9.1 Functional description

The primary function of the Hardcopy daemon is to print sequential text files on hardcopy devices. Some of the subfunctions it performs are as follows:

- It appends classification markings to the top and bottom of each printed page.
- It appends a disclaimer page or disclosure form at the beginning of any printed text file that requires it.
- If a printer is inoperative, or is of insufficient security level, it prints on an alternate printer specified for that printer, or on the TENEX line printer if this printer is already being used as backup for printing the file.
- For particular classifications, it also marks the printed document with a SIGMA sequence number, unique to that document, which it also reports to a logging file.

4.12.9.2 Daemon operation

The Hardcopy daemon is designed as a set of printer-driving processes, in which each process outputs files to a specific printer, along with a superior monitoring process to detect the occurrence of hung printer processes. Each printer process maintains a state determining which printer it is driving and which terminal line corresponds to that printer.

Each printer is connected to its own special terminal line, which is driven by device-dependent monitor code and a special PDP-11 processor front-end to implement the printer control discipline. As part of that discipline, should the physical printer become unable to proceed (because it is out of paper, powered down, or some other similar condition), any output in progress will be suspended (i.e., any process doing output to the printer will itself be suspended) until the device is ready to continue.

The monitoring process maintains a communication page in its address space, which it shares with all of its inferiors. To detect whether a printer process is not progressing, the controller resets a cell corresponding to the printer process. If this cell is not set the next time the controller inspects the cell (two minutes later), the printer process is considered hung and is killed and re-created with its internal state changed to cause its printed output to be rerouted to a backup printer, or queued for printing on the TENEX printer if the process had been doing backup printing when it hung.

Every printing process sets its cell in the shared page at least once a minute to show that it is not hung.

Each printing process "n" prints files <HARDCOPY>LPTn.xxx and <HARDCOPY>BACKUPn.xxx, where the LPTn files are for primary printing, the BACKUPn files are files that have been renamed for backup printing on printer n, "n" is the decimal representation of the printer number (a nonnegative integer), and "xxx" is a string indicating the user for whom the printout is destined. Each process looks for these files at least once every minute.

The printer number is assigned to a printing process by the monitoring process when the process is created; the particular correspondence between processes, printer numbers, terminal line numbers, security levels, and backup printer numbers is controlled by another sequential text file, <SIGMA-LOCAL-STATE>HRDCPY.PARMSFILE, which the controlling process reads at the time it is started. This file specifies the number of printers, and, for each printer, its SIGMA printer number, its TENEX terminal line number, its security level, and its backup printer's SIGMA printer number. Thus the configuration of printers is also under installation control.

To allow the direct use of the TENEX line printer for, say, an office that does not yet have its own physical printer, SIGMA printer 0 is treated as the TENEX line printer and is assumed to have no terminal line number, unlimited security, and no backup printer.

Files at a security level above the maximum level for a printer will not be printed by the process: if a file is LPTn.xxx, it will be renamed BACKUPm.xxx, where m is the backup printer number for printer n. If the file is BACKUPn.xxx, it will be printed by printer process n on the TENEX line printer.

The markings appended to each page, and the disclosure form or disclaimer form, are contained in sequential text files in directory SIGMA-LOCAL-STATE, named as follows: UNCLAS, CONFID, SECRET, and TOPSEC (corresponding to the classifications unclassified, confidential, secret, and top secret). The extension part of the file name indicates the type of marking: TOP indicates a top-of-page marking, BOT indicates a bottom-of-page marking, and FRM denotes a disclosure form. The disclaimer page is contained (for all classifications that do not have a disclosure form) in the file DISCLAIMER.TEXT. Thus the precise form of the markings is under the control of the installation running the daemon, rather than being fixed in code.

In addition to the top and bottom markings, the first page of output (either the disclaimer form or a disclosure form) will include markings at the top (directly below the top markings) indicating the user for whom the output is intended.

If there is no file for a particular marking, the daemon will not print anything where the marking would go, with the exception of the disclosure forms, for which either the disclaimer form will be printed or no disclaimer page will be appended.

For those classifications that require special handling, the top and bottom markings and the disclosure forms may include one place each in the text for the insertion of a 9-digit SIGMA sequence number. The place in the text is denoted by a tilde ("~") followed by nine characters (values of the characters are not important). The printing process will, if it needs to print a disclosure form for a particular classification, insert also a unique sequence number (beginning from 1) in that place in the marking. The sequence number will be determined from the value of the first byte of the file <SIGMA-LOCAL-STATE>HRDCPY.SEQNR, and the byte in the file will be incremented.

If there is no disclosure form for the classification, no sequence number will be printed or generated.

When a document is printed with such a sequence number, the numbers of the document and the printer are reported to the SIGMA accountability file.

4.12.10 Maintenance Daemon

4.12.10.1 Functional description

The sole function of the Maintenance daemon is to periodically expunge the database and daemon login directories of deleted files, when the free file space falls below a certain threshold.

4.12.10.2 Rationale

Files need to be expunged from the database and daemon login directories because of the rapid buildup of deleted message, folder, and working files during the creation and updating of message and folder files. Since deleted files still take disk and directory space, a condition will eventually occur in which a daemon process will be unable to create a new file, or modify an existing file, in a database directory because of insufficient space. In order to release the space taken by these files, a process must expunge the directory, recovering all disk and directory space occupied by deleted files within the directory.

Since the expunge operation is relatively expensive in process execution time, a daemon process performs this operation only when it is unable to open a file for write on any of its database directories. However, the execution of the expunge operation on any or all of a daemon's directories does not guarantee that the system will not see a "no room" condition, that is, a condition in which the file system has insufficient free disk space (the threshold number of pages is a fixed parameter for the operating system, usually set to 1000 pages). No single daemon may be able to alleviate this condition (e.g., the Message daemon will not be able to alleviate a "no room" condition caused by the buildup of deleted folder files).

To minimize the need for expunges in the other daemons, and to address the problem of a "no room" situation which a daemon is not able to repair on its own, the Maintenance daemon performs expunges on all daemon database directories when the free file space becomes lower than a preset minimum, empirically determined from the observed load SIGMA places on the file system.

4.12.10.3 Daemon operation

The Maintenance daemon, in a 15-minute cycle, operates as follows:

- It determines whether the file space is low; if it is not low, the daemon does nothing, skipping all of the following steps. Low file space is set in the code to be 30000 pages.

- It connects in turn to all of the message, folder, archive, and daemon directories, and in each directory, it expunges the deleted files.

The daemon keeps a record in its log of the results of each of its cycles in which file space was found to be low; the record includes the space found and the space recovered as a result of the expunges.

4.13 PACKAGES

4.13.1 Error Package

4.13.1.1 Motivating considerations

This package was developed especially for the daemons, for handling unusual circumstances and error conditions. Before the error package was produced, the following situation prevailed:

- Only two "standard" actions could be taken in response to an error condition: tracing and the invocation of the SYSTEMERROR subroutine call.

- All tracing could ordinarily be done only to a single file for any particular process; the resulting file was often very hard to use, cluttered, and very large: the small useful part was buried under mountains of other data.

- Logging (i.e., the keeping of records for some external use, such as perusal by administrative personnel of some process's actions) was ad hoc.

- There was no convenient way to notify the operator of any anomalous conditions that needed attention.

- The standard error action, SYSTEMERROR, did not provide enough information to the programmer, even in the traces, to determine the real cause of the problem; sometimes it could be found only by examination of the core image, which was not saved.

The result was that, at least for the daemons, it was hard for the operator (or almost anyone else) to determine what had happened when an error occurred.

Thus, it became apparent that a package that would provide a more comprehensive set of standard trace functions and standard error actions was needed.

4.13.1.2 Error package general description

The error package is a collection of BLISS routines and macros that

- provide tracing, logging, and error reporting functions to various files according to a standardized policy,

- provide a set of standardized actions for use under different error and nonerror conditions,

- provide a standardized set of tracing policies and error actions, and

- allow nonstandard tracing and error actions at small additional programmer effort.

The error package is implemented as two BLISS modules and three collections of BLISS macros. The BLISS modules supply the basic functional services, as follows:

MP	The multiprint module supports the output of text to one or more destinations, using pointers to arrays of destinations as the output specifications.
ER	The error-handling module supports a standard set of error tracing and reporting policies. The principal routine, ERX, accepts as parameters the multiprint destination (i.e., the files which will receive the traced information), an arbitrarily long list of items to be traced and format codes for their tracing, and a similar list of actions to be taken to process the tracing request. The latter may include tracing, but may also be any BLISS statement.

The macro collections build upon the facilities supplied by MP and ER, providing a convenient set of calls for the programmers:

RE.REQ	These macros allow the specification of actions to take under certain conditions or unconditionally, and the specification of items to be traced in the error logs. The action(s) to be taken are specified as an arbitrary BLISS statement, which is expanded so as to be executed after the tracing of specified information to the error logs. Also in these macros is a set of useful actions to take, specified as the macros RECORD, REPORT, RETIRE, and RESTART.
TR.REQ	These macros support tracing of up to eight arbitrary words of data, together with a string of text, to a default trace file, as octal numbers, in a standard format.
LO.REQ	These macros support tracing of words or strings, in specifiable formats, to a set of logging files. The format for the text can be made much prettier, since the interpretation of the data is not limited to octal numbers.

4.13.1.3 MP: Multiprint output package

The purpose of this package is to support printing of error and log files. The Multiprint package has two requirements. First, it allows the specification of output to several JFNs, instead of the one provided in the normal JSYSs. Second, it provides the high-level routines required by known logging and error functions.

Multiprint file designators (MPFDs)

A multiprint file designator is in the form of a PLIT. The -1st entry is the number of output designators. Each entry must be a JFN, a byte pointer, or the SIXBIT code "CIRCLE." In the latter case, the output is sent to the circular buffer (described below in section 4.13.1.4). Note that the PLIT must be in read/write storage as it may be modified.

There are two legal cases for byte pointers. If the byte pointer specifies a byte size of 7, then the literal byte pointer is used as the destination for the output. If the byte pointer specifies 18- or 36-bit bytes, then it is evaluated as an indirect reference to the actual byte pointer. The evaluation of these pointers is done only once. Multiple levels of indirection are not allowed. Thus, this facility only implements a deferred loading, *not* an indirect reference.

Services provided

The MP package provides a number of text output services similar to those provided by the uniform TENEX output JSYSs (**BOUT**, **NOUT**, **ODTIM**, etc.), but produces output for all those destinations specified by a MPFD rather than a single TENEX output designator. The services are grouped into three levels, based on the complexity of the output generated:

1. The *low-level* routines provide services directly analogous to the TENEX JSYSs.

2. The *medium-level* routines provide such additional services as symbolic address printing, load average and a file's status.

3. The *high-level* routines produce large, complex printouts, such as the file or memory status of a job, the subroutine call stack trace, or the dump of the circular trace buffer.

4.13.1.4 ER: The error-handling package

The error-handling package consists of two parts. The first is concerned with the maintenance of multiprint file designators and other state information required by the error package. The second part is the error processor which takes event-processing descriptions and processes them.

Trace files

The error package maintains four tracing files, one for each of the levels of detail desired in tracing. The files all have a common name, and an extension indicating the intended tracing level, as follows:

Trace level	File extension	Intended for
Tracing	.TRACE	Application programmer debugging
Logging	.LOG	Operations staff monitoring
Short error	.SHORT-ERROR-LOG	System programmer information
Detailed error	.ERROR-LOG	Application programmer debugging

These files' JFNs are maintained in global cells in module ERGLOB.

Additionally, there is a one-page circular buffer maintained, which contains the last 2560 characters written to the tracing file. This buffer is copied to the error log file in the event of a serious error.

For efficiency, the production versions of the user job and daemons do not maintain certain of the above files: the production user-job omits tracing to the trace file and circular buffer; the production daemons omit tracing to the trace file but maintain tracing to the circular buffer. This is a tradeoff between performance in production versions and recording sufficient information in the case of errors to be able to isolate and correct the problem.

Trace policy

The module ERP contains a set of PLITs that define a standard error tracing policy. Specifically, the PLITs point to global arrays which contain five MPFDs, one for each of the five tracing policies. The PLITs in ERP are used to initialize the tracing MPFDs to correspond to the tracing policy, as follows:

Trace MPFD	Output destinations
TRACE	Tracing output to the trace file and the circular buffer.
LOG	Output to log file and the tracing destinations.
SHORT	Output to the short error log and the tracing destinations.
HEADER	Output to both error logs and the tracing destinations.
ERROR	Output to the long error log only.

Event processing

The error package supports the processing, in a number of standardized ways, of any "interesting" event that needs to be logged or which represents an error or anomalous situation. The processing of such an event is specified by means of an event description, in three parts:

1. *Destination determination.* This is an MPFD specifying where information will be recorded during processing. *It is usually one of the five "standard" MPFDs, described above.*

2. *Control procedure.* This is the sequence of steps to be executed in processing the event. It is a PLIT of event-processing descriptors, interpreted by the event-processing routine (routine ERX, module ERX). The descriptors include items to trace (e.g., list of files, job process structure, process accumulator values), and other actions to take (e.g., tell the CCP operator, save the process state in a file, or stop the daemon for an interval).

3. *Supplementary data.* This is the specification of data values and formats to be traced, in addition to any that may be traced by the control procedure. The intent is that the control procedure will be a standard one for a large class of events (e.g., RETIRE), whereas these data are to be recorded for a particular invocation of the procedure. The processing of the supplementary data is itself specified in the control procedure (i.e., it is not a "default" action of the interpreter).

To provide process-specific processing for certain classes of standard actions, the error package supports the specification of routines to be called to achieve certain steps in the event processing. The application process code may call routine ERX.SET to set them, or may allow default routines from the error package itself to be used. Functions which the application may specify to ER are:

STOP	Halt the process for a specified amount of time.
CLEANUP	This routine will be called to do any cleanup tasks (e.g., closing open files, setting or clearing status bits) needed before killing the process.

TELL	This routine is called to communicate a specified text string to the CCP operator process.
SHOW	This routine is called to set a status code in a global location which is then available to various monitoring programs to display the process's current status.

4.13.1.5 RE.REQ: The error-reporting macros

The RE package provides a set of error-monitoring constructs which allow a programmer to specify conditions under which error handling is necessary, as well as a group of standard error-handling procedures.

Error-monitoring constructs

Two forms of error-monitoring constructs are supplied in RE: the conditional form, WHEN, specifies an error condition when a specific condition is satisfied; the unconditional form, OOPS, causes error processing whenever it is executed. In the BLISS source, their formats are

```
OOPS(<action>, '<string>');
OOPSn(<action>, '<string>', <expr1>, ... , <exprn>);

WHEN(<boolean>, <action>, '<string>');
WHENn(<boolean>, <action>, '<string>', <expr1>, ... , <exprn>);
```

where *n* can be 1 through 4.

The parameters in the above formats have the following meanings:

- *<boolean>* specifies an arbitrary Boolean expression which describes conditions under which error-processing is warrented.
- *<action>* is a BLISS statement (which may be null) which will be executed after the "standard" error handling is performed (see below).
- *'<string>'* is an arbitrary text string which will be included in the error report.
- *<expr$_n$>* are BLISS expressions whose values will be printed in the error report.

The basic form of what each macro prints, and in which files, is specified in RE.REQ and ERX.BLI. This is in essence a policy determination done with a set of macros and PLITs. For instance, OOPS first prints three standard lines and one line identifying the error source (the arguments to OOPS) in the Trace and the Short and Long Error Logs.

Error-handling procedures

These procedures, defined as BLISS macros, define standard error-handling procedures, and are normally expected to be used as the *<action>* for one of the error-monitoring constructs described above. They span a range of severity levels:

RECORD	A condition has occurred which is not normally expected; however, it is sufficient to make a short entry in the error logs and proceed. This recording procedure may also be used to record information when recovering from errors.

REPORT The condition is not immediately fatal, but requires operator intervention in order for it to be fixed (e.g., a bad file is encountered). This action records the problems and notifies the operator. As with RECORD, processing continues.

RESTART A serious condition has occurred where the only recovery is to restart the process from scratch. This implies a strong belief that restart will in fact solve the problem. Pertinent data are written into the error logs, and the process is halted. If no operator intervention occurs within a prespecified time interval, the process is restarted.

RETIRE A serious condition has occurred where no automatic recovery is possible. The operator must investigate the errant process before restarting can be attempted. All pertinent data are written into the long error log, an appropriate subset is written into the short error log, and the process is halted. The process will not restart automatically; the operator must explicitly restart the process after determining and correcting the source of the problem.

While these procedures are normally used as the *<action>* of an error-monitoring construct, other valid BLISS statements can also be used, and these standard error procedures can be combined with other actions, e.g.,

```
WHEN1(.PARM LSS 0,
      (RECORD; RETURN FALSE),
      'Parameter out of range',
      .PARM);
```

4.13.1.6 TR.REQ: The tracing macros

This file defines a collection of macros which implement the proper calls on the multiprint (MP) package for simple tracing of octal values and strings. The trace macros have the following form in the source:

```
TRACE('identifying string');
TRACn('identifying string', <expr1>, ... ,<exprn>);  {0<n<11}
TRACP('identifying string', <array pointer>);
TRACZ('identifying string', <string pointer>);
```

Each of these macros prints the identifying string, with the latter three calls printing a counted number of expression values, the elements of an array in PLIT format, and a string identified by its pointer, respectively. Calls on these macros print only in the Trace file.

4.13.1.7 LO.REQ: The logging macros

This file defines a set of macros which produce nicely formatted output suitable for administrative logs and other nontechnical output. The source format for the logging macros is:

```
LOG('<string>');

LOGn('<string1>',<type1>,<data1>,
          ...,
          '<stringn>',<typen>,<datan>);
```

where *n* can be 1 through 5.

The <type> strings determine how to interpret each <data> value, according to the following encoding:

'OCT'	Print as octal number
'DEC'	Print as decimal number
'STR'	Print ASCIZ string pointed to by value
'SIX'	Print as SIXBIT string
'DIR'	Print value translated to directory string

As an example, the logging specification

```
    LOG3('Citation received for user ', 'DIR', .USERID,
         ', citation count = ', 'DEC', .CITCNT,
         '?M?J?IMSGID = ', 'STR', .MIDSTR);
```

would produce output something like the following in the logging destinations:

```
  Citation received for user J301, citation count = 5
         MSGID = J301 091545Z JUL 79
```

4.13.2 Text Package

The Text Package is a set of routines that support paged text. It was designed to improve the performance of earlier text-handling schemes, while still providing compatibility with those schemes. The Text Package routines provide the commonly needed text-processing functions (including search) suitable for the various SIGMA applications.

4.13.2.1 Motivational considerations

Text size. SIGMA processes handle objects that often include massive amounts of text, sometimes more text than will fit into a process address space (e.g., folders often have this characteristic). Thus, text from a file must be kept stored in the file, referenced by pointers of some kind, and actually accessed only as necessary, rather than being read into a process all at once.

Text access. Earlier text-handling schemes in SIGMA had a serious drawback because the text was stored in the address space of each SIGMA process. This required either copying of text from one process to another, or clumsy mechanisms to allow ad hoc sharing between processes. The Text Package provides uniform, global access to text to any of the processes in the SIGMA environment without copying.

Compatibility with previous schemes. Previous text-handling schemes used a half-word quantity to point to a piece of text, generally by specifying the address of the start of the text in the process address space. The Text Package uses a full-word quantity for this same purpose, but allows the use of a half-word portion of the quantity, together with a JFN on the text file, to be compatible with the half-word schemes.

Compatibility with other structures in a file. For files that have text (messages, folders, etc.), additional data are often also present (e.g., messagette structure for messages, entry data for folders). To allow this data to continue to exist with minimum modification, the text needs to be put somewhere else. Since these data are built in file pages 511 and below, the Text Package portion of a file is designed to reside in pages 512 and up. These pages are in the "long" portion of files, and they do not interfere with the data stored in the "regular" or "short" portion of the files (for further description of the TENEX file structure, see section 4.3.3, page 4-8).

4.13.2.2 Design overview

In SIGMA, text resides in pages of the file not normally accessed by a process (pages 512 on up), the "long" portion. The Text Package uses a process-specified working area for mapping paged text into the process address space and for various utility functions. The package routines automatically map pages into or out of the working area as necessary to perform the functions required. The package requires no other work space (other than fixed global and local storage). Thus no dynamically allocated address space storage is required for handling arbitrarily varying amounts of text, beyond that required by the application for storing text identifiers (see below).

4.13.2.3 Detailed design description

Text identifiers

A process in SIGMA references text managed by the Text Package by means of a *text identifier*, or *TID*, a unique handle on a piece of data managed by the Text Package. A TID is a 36-bit quantity, and may take one of three forms:

1. *Simple*: In this form, the left half contains the JFN of a valid text file, and the right half contains a pointer to a descriptor within that text file naming a particular text object (called a *text name* or *T-name*).

2. *Bound*: This form, which is used internally by all Text Package routines, has a T-name in the right half as in the simple form above, but the left half contains descriptive information extracted from the referenced text object which allows for efficient processing: the JFN, length, and type of the object.

3. *Chained*: This temporary form takes on the format of a PDP-10 byte pointer, where the subject of the byte pointer is one of these three forms (i.e., multiple indirection is allowed). When a TID of this form is encountered, the chain of indirections is followed to the end, then the bound form of the final TID is substituted for all of the indirect references along the chain.

In the first two cases above (the common ones), the right half of the TID, the T-name, will uniquely identify a text object provided its containing file is known. Since this is the case for most SIGMA objects, this Text Package naming scheme is compatible with previous implementations where only a half-word was allotted for a text descriptor.

Text segments

The text space of a file is divided into 512-page sections known as *text segments*, or *T-segments*. Each T-segment is self-contained; it contains all data needed for manipulating text within it and contains no references to other T-segments.

A T-segment is composed of three parts: the *text header portion*, the *text storage portion*, and the *free storage portion*. These are described below.

Text header portion

The text header portion takes up the first 64 pages of the T-segment. It is composed of two-word entries called *text headers*, each of which describes one text object. A T-name maps to exactly one text header through a trivial calculation. Each text header contains the following descriptive information:

- A reference count keeping track of the number of references to this text object.

- A pointer to the text value (see below).

- An indication of the text object's type (simple or complex, see below) and content (text or binary).

- The length of the object (bytes for text data, words for binary).

- A hash value for text data, for search efficiency.

Text headers are allocated on a fixed-block allocation scheme, with a free list to hold deallocated headers.

Text value portion

The text value portion composes the 447 pages of a T-segment following the text header portion, and contains the value of all text objects within the T-segment. Each text value can be one of two forms, *simple* or *complex*:

- A simple data value is a contiguous set of words containing the text or binary data comprising the value of the object. If the object is a text object, the words are interpreted as a sequence of left-adjusted 7-bit ASCII bytes.

- A complex object represents either a substring (*subvalue*) of another object, or a concatenation (*supervalue*) of substrings of one or more other objects. It contains a counted list of two-word descriptors, each describing a substring of some other object (including the entire object). This complex representation allows substring and concatenation operations to be performed simply by the creation of new descriptors rather than the copying of the data.

Free storage portion

This is the last page (page 511) of the T-segment, containing free storage headers, used by the Text Package page management routines. Free storage consists of varying-size blocks with a buddy system. Additionally, the page includes two words used for locking of the segment when Text Packages in different processes need concurrent access to the segment, and the root pointer for the text header free block list.

4.13.2.4 Summary of text package routines

Following is a brief outline of the routines provided by the Text Package, grouped into categories of related functions.

- The *object creation* routines provide for creating text objects from a single byte, a string of characters, or a block of data, returning the TID for the object created.

- The *access* routines provide for outputting the text or data of a TID, either whole or a contiguous subpart, to an area of the process's address space or to an open file.

- The *object manipulation* routines provide facilities for: release of a TID; copying of a text object, to the same or a different text file; extraction of a contiguous subpart of another text object; concatenation of two text objects; insertion of text into an existing text object.

- The *searching* routines provide mechanisms for searching and testing certain attributes of strings.

- The *setup and file-handling* routines handle initialization and termination of access to the Text Package, initialization of text files, and copying of text files.

- The *sequential file operation* routines provide for sequential reading and writing text objects to and from sequential files.

- The *byte pointer utility* routines allow incrementing and decrementing of PDP-10 byte pointers and the copying of strings from one place in memory to another.

4.13.3 Message and Folder Directory Access Packages

Messages and Folders are maintained with the aid of two packages, MD and AFOLAC, that control access to directory files for the Message and Folder databases, respectively. They provide analogous functions in each database and will be described together in what follows.

The database directory files (or directory files) contain status and location information for each object in the database. The purpose of the directory access package is to control all accesses to the directory files, so as to coordinate their reading and concurrent updating by asynchronous SIGMA processes in both the user job and the daemons.

The directory access packages provide routines for reading entries in the directory files and for changing entries in those files (see section 4.11.1, page 4-65, and section 4.11.2, page 4-69, for descriptions of these entries).

The entry information provides any process using the directory access package fast access to the objects in the database. The coordinated updating portions of the directory access packages allow daemon processes to keep the access information current without interfering excessively with the activities of processes merely reading entries to access objects.

4.13.3.1 Directory access package functions

The directory access packages provide routines that allow a process to search for an entry for an object, to make a new entry for an object, to change the entry for an object, and to delete entries for objects. Additionally, the AFOLAC package provides specialized routines for maintaining the busy state bit for specific folders. This function is not provided for the MD package as the busy bits for a message are contained in the message and are maintained by MSGMOD.

4.13.3.2 Directory access package access coordination

The MD package uses lock words in the directory access files for messages to inhibit updating of entries while a read is in progress and vice versa. The AFOLAC package uses a lock file, open for write only while a directory file access is in progress, for the same purpose. Since accesses to the directory access files are very fast, the directory entries are locked only for very small intervals, only slightly greater than the duration needed to access the one-word entries.

4.13.4 Lexicon Package

4.13.4.1 Design

This set of routines and Bliss macros maintains, within a file, mappings from numbers or text strings to and from numbers or text strings. The implementation uses AVL trees for efficient search for both text and numbers.

Lexicon files may be built by the application program, or may be constructed by a programmer with the aid of a utility OFFLINE. This utility allows the programmer to specify in a text file the mappings desired between numbers and/or strings and to build corresponding binary files containing the tree structures required for the use of the Lexicon Package.

A lexicon file is read into the process address space of the process desiring to use it. The process may simply read the mappings (i.e., perform only searches), or, if the file is opened for reading and writing, the process may dynamically change the file (i.e., enter new mappings, delete old ones).

The routines and Bliss macros provide the following services:

- enter a mapping,
- delete a mapping,
- examine a particular mapping,
- search for particular mapping(s) on the basis of target value(s) (this is discussed in more detail below),
- copy lexicon files,
- various other file-related operations: open, close, initialize, or obtain or modify certain file attributes.

4.13.4.2 Lexicon entries

An entry in a lexicon file is a mapping between a target and a definition. The target and the definition may each be a text string, a decimal integer, or an octal integer (the two kinds of integers have the same range of values but are conceptually distinct within the package). Additionally, each target and each definition has an associated datum value, which is an integer between 0 and 262143 inclusive (i.e., 18 bits of numeric information).

4.13.4.3 Searching operations

The principal use of the Lexicon Package is for searching for possible matches to a text string or, sometimes, for a particular numeric value. The search operations allow for finding certain classes of partial matches, including certain kinds of spelling corrected matches. The specific searching operations are as follows:

1. Exact match: Look for an exact copy of the target string in the file.

2. Case-independent match: Look for a copy, but ignore the case shift of letters.

3. Partial match at the beginning: Look for a copy of the target string as the beginning part of an entry in the file.

4. Spelling corrected match: Look for an entry that matches if one of the following kinds of spelling errors is assumed to have been made to the target: a character has been doubled, a character has been omitted, or two adjacent characters have been transposed.

5. Any combination of 2, 3, and 4 together.

6. Enumeration of all strings (or numbers) in the lexicon.

Once a target value is found in the lexicon file, the package provides macros to do the following:

Examine the entry.
: Any part of the entry may be examined: the target value, the target datum, the definition value, the definition datum, and the type of the definition.

Find the next match.
: The next entry that matches the search criteria that found the current entry will be found.

Find the next definition value.
: The target value found may have more than one definition value mapped to it. The next definition for the target is found, if it exists; otherwise, the next matching target (if any) is found instead.

4.13.4.4 Uses in SIGMA

Command Language Processor

The Command Language Processor (CLP) makes extensive use of the Lexicon Package to support the command recognition and spelling correction features. The lexicon files that the CLP uses in fact constitute the command table for the CLP. In principle, to provide a different set of commands (say, in a different language, such as French, or with possibly different semantic interpretations in SIGMA), all that is needed is a new set of lexicon files.

Reception Daemon

The distribution of AUTODIN messages is made with the aid of a lexicon file that maps various LDMX-supplied destination codes (strings) to the names of users or offices in the SIGMA system.

User Error-Strings and Error Messages

Another lexicon file, USER-ERRORS.EXPANSION-TABLE, is used throughout SIGMA to map short strings (i.e., strings which fit into a single machine word) to longer text strings intended for reading by a user or other person. The intent is to allow the developers of SIGMA to specify a particular error in symbolic form rather than in, say, a numeric code. The principal cost of this, the search for matching the symbolic error code, is minimized by the use of an efficient tree structure for storing the lexicon coupled with an efficient tree search algorithm. Since the search for an error string for a given code occurs only infrequently (when there is some kind of error, typically), the cost is thought to be reasonable.

4.13.5 The Queue Package

4.13.5.1 Introduction

The Queue Package consists of a collection of routines which are used to create, initialize, and access standard queues that need a multiwriter single-reader discipline. The term "standard" is used to differentiate the queues supported by the queue package from other queues that were implemented independently under various queue disciplines. The Queue Package is loaded by both the reader and writer of a queue. A process need only load the Queue Package once regardless of how many queues it deals with.

4.13.5.2 Queue structure

Queues are intended to exist forever once they are created (even when SIGMA is not in operation). Therefore they are implemented as permanent TENEX files. Queues are divided logically into two parts: queue proper and overflow. The queue-proper portion of the file is mapped by all processes using a particular queue. The remaining file space forms an overflow area.

The overflow area is the "rest" of the queue file beyond the queue-proper portion. It is treated as a sequential file region and all overflow blocks are written there. Enqueing into the overflow area is transparent to the user. The overflow routine is called from the normal enqueue routine when the overflow condition is detected. From that point on all blocks are enqueued into the overflow region until it can be emptied by enqueing into the queue-proper region. Enqueing from overflow into queue-proper is initiated from the dequeue routine any time it determines that there is approximately one page of free storage available.

4.13.5.3 Block structure

Figure 4-16 shows the appearance of a block of enqueued information in the queue.

4.13.5.4 Queue state information

The queue header occupies the initial locations of page zero of the queue. The following information will be kept in the queue header.

LOCK -	lock for writers
READER POINTER -	reader pointer
WRITER POINTER -	writer pointer
OVERFLOW READER POINTER	
OVERFLOW WRITER POINTER	
QUEUE OWNER	

```
                    +----------+----------+
                    | COUNT    | NLINK    |
                    +----------+----------+
                    |                     |
                    |                     |

                             DATA

                    |                     |
                    |                     |
                    +----------+----------+
                    | BLINK    | LINKID   |
                    +----------+----------+

            where   COUNT - block size in words (including control
                            structure words)

                    NLINK - link to next block in this list

                    BLINK - link to header word of this block
                            (used for consistency checking)

                    LINKID - value on which current list is linked
                            e.g. a particular file ID
```

Figure 4-16: Queue block structure

LAST ENQer	
RECORD -	last returned address from queue
RECORD COUNT -	number of times last block was requested (and not DEQ'd)
OVERFLOW FLAG -	TRUE => nonempty overflow area
CITATION QUEUE FLAG -	TRUE => citation daemon queue
VIRTUAL COUNT -	number of virtual QFIRST's done
VIRTUAL RECORD -	last returned address from QQQVFIRST
SIZE OF QUEUE IN PAGES	
LOCKER -	last process to lock queue
CONSISTENCY WORD	

4.13.5.5 Special queue package functions

The queue package supplies all the obvious routines for handling queues and accounting for errors with respect to the error package. In addition, there are several functions that have been implemented for specific SIGMA considerations.

4.13.5.5.1 Automatic DEQ'ing of blocks

Every time a block is requested, the RECORD COUNT in the queue header is incremented. Should a block be requested for the third time, it is automatically dumped into the error log and DEQ'd. This procedure prevents a consumer from looping on bad blocks.

Just before DEQ'ing the bad block, the queue package will call a routine that has been (optionally) supplied by the consumer process. In the daemon case, this "last gasp" routine is used to clean up any files or busy bits that may be relevant to this block.

4.13.5.5.2 Exception handling

In addition to normal error handling and automatic DEQ'ing, the consumer can move entries from the queue to a separate EXCEPTION file. A stand-alone utility called EXCEPT provides many functions for viewing and manipulating the queue entries in this file. In particular, entries can be re-ENQ'd to a daemon queue for processing, presumably when the problems that caused the entry to put in the EXCEPTION file are resolved. This process was designed specifically for the Citation daemon but is available to any consumer.

4.13.5.5.3 Linking queue entries

In its normal mode the queue package presents a first-in first-out (FIFO) discipline. However, due to the efficiency considerations needed in citation handling (section 4.12.6.3, page 4-82) a facility to link queue entries is provided. Thus, when ENQ'ing an entry, the caller may specify a LINKID. If the LINKID is non-zero, the ENQ routine puts the entry at the end of the queue, finds the last queue entry that has the same LINKID by looking in a separate LINKID file (used expressly for this purpose), puts the address of the new entry in the NLINK field, and puts the address of the newly ENQ'd entry back into the LINKID file as the last entry for that ID. Thus, the queue consumer has the option of accessing elements in queue order, link order, or some combination of both.

4.13.6 Data Collection Facility (DCF)

An important aspect of the MME project is to gather data about the use of SIGMA at CINCPAC. The SIGMA user job is instrumented to generate dynamic data for this purpose for later analysis by the Mitre Corporation. The Data Collection Facility (DCF) is designed to give the FM and CLP a set of routines for writing relevant data to an external file. Points are written for specific predesignated events, execution of functions keys, errors, commands, etc. The exact information in each DCF point varies with the type of event, but basically, each point has the following information: the type of point, time of generation, load average, the ID of the objects referenced or created, the command involved. A single user command may generate several points.

SIGMA does nothing with the DCF file produced. We do, however, have a utility called DCF which prints out a DCF file in a semi-readable way.

4.14 UTILITIES

A number of SIGMA-specific utility programs were developed to aid in the general operation of SIGMA. These were programs started up from the TENEX Exec from standard TENEX terminals by system programmers or operators at appropriate times. These utilities complemented the large set of similar programs provided with TENEX.

4.14.1 EXCEPT

EXCEPT provides the operator with a means for disposing of undeliverable citations. The citations are saved by the citation daemon on the file EXCEPTION.CITATION in <DAEMON-LOCAL-STATE>. This utility reads the file and allows the operator to select sets of citations to be requeued on the Citation queue or deleted from the EXCEPTION file.

Criteria for selection are:

- before a given date and time (inclusive),

- after a given date and time (inclusive),

- pertaining to a particular folder,

- pertaining to a particular message,

- complements of the above,

- citations already deleted.

The selection of a set of citations is made by restricting or augmenting the collection of citations in the file, using the criteria above. The set finally selected may then all be requeued, deleted, or undeleted (i.e., unmarked for removal from the file). The actual removal from the file of deleted and requeued citations is done as a separate explicit command.

4.14.2 FCHECK

FCHECK provides the operator with a means to check the existence of critical files, and to verify the consistency of the folder database. Each of these functions may be selected independently. These checks are made over all the files in the folder database directories and all of the entries in the folder database index file. See section 4.11.2, page 4-69, for details of this file.

The existence check is driven by a file with a list of file names and attributes to be checked. Attributes that can be checked are:

- file is openable,

- file is a share save file,

- file is a lexicon.

The folder database checks are for:

- the folder can be opened,

- the folder contains a valid verification word,

- the folder identification agrees with the (TENEX) name of the folder file,

- there is agreement between the folder database index file and the folder file regarding:

  1. folder ownership,

  2. folder existence in the particular TENEX directory.

4.14.3 FLAGS

This utility and SSO (see section 4.14.4, page 4-106) provide for the setting of certain attributes relating to specific terminal lines. The attributes settable by FLAGS are stored in the file <SIGMA-LOCAL-STATE>FLAGS.SIGMA.

The attributes and range of values are:

- Allow SIGMA, 0 or 1: SIGMA will run on terminal line only if this is set.

- Indirect, 0 or 1: SIGMA will use a terminal assigned to the job in place of the controlling terminal.

- Function key layout, 0 to 8: Any of eight predefined layouts for the function keys may be specified. CINCPAC used layout number 1.

- 2-bytes checksum, 0 or 1: In the transition between one- and two-byte checksum terminals, this attribute determined which protocol SIGMA should use with the terminal.

- HP terminal traces, 0 or 1: This flag affects only the development version of SIGMA, XSIGMA. If 1, a new version of the file {USER-STATE/SIGMA}.TRACE will be created and will contain a printout, 1 per line, of each terminal dispatch and notice with each byte of the dispatch and notice given as an octal value. The octal sequence number of each dispatch and notice is also traced.

- Data collection, 0 or 1: If 1, SIGMA will enter information for the session into data collection files (see section 4.13.6, page 4-104, for details).

- Miscellaneous: 0 through 31. (This is known as "Tugender," for historical reasons). This value is simple five bits settable for experimental features useful in program development. Production SIGMA ignores them.

4.14.4 SSO

SSO (which stands for "System Security Officer") allows the SIGMA operator to add or delete specific users' access to SIGMA, or to change their access rights; and to change certain SIGMA access rights associated with specific terminal lines (see section 4.14.3, page 4-106, for the other settable characteristics for terminal lines).

For a given user, the SIGMA operator may do any of the following:

- add or remove that user's capability to use SIGMA;
- change the user's security level to one of unclassified, confidential, secret, or top secret;
- set the user's release authority concerning informal notes, formal memoranda, or formal AUTODIN messages;
- set the user's name to be either a title or an office code;
- set the user's experience level to be one of novice, intermediate, or expert (expert was later dropped);
- set up a real or a fake Pending file for the user (a fake Pending file was set for a user who did not have access to SIGMA, but who would appear as an addressee in a distribution list);
- configure the user's real Pending file according to various groupings of security levels (see Part 4, section 3.3.2.16, page 3-21).

For a given terminal line, the SIGMA operator may do any of the following:

- set the maximum security level of the line;
- set the maximum release authority for the line;
- set the number of the hardcopy printer associated with the line.

The SIGMA operator may also obtain a file listing the current settings of the users' and terminal lines' parameters.

SSO is intended to control parameters associated with specific users' and terminal lines' access to SIGMA to aid in implementing the security requirements concerning access of classified information by certain users and within certain environments. Additionally, it aids in controlling parameters associated with users' level of experience with SIGMA.

4.14.5 MEDIT

This program provides the SIGMA operator with a means to examine and/or change the entry for a message file in the message index. The entry attributes changeable are (see section 4.11.1, page 4-65, for details):

- the message state,
- the message on-line-directory,

- the message archive-directory,

- the message security level.

4.14.6 MSCAN

MSCAN provides the SIGMA operator with a means to check the consistency of the on-line messages in the message database. The program checks all files in the message database directories (SIGMA-MESSAGE1, SIGMA-MESSAGE2, etc.) against the message database index file for agreement.

The program reports the following kinds of discrepancies:

- Files with names not translatable to message ID's.

- Files with names translatable to message ID's, but the entry for that ID disagrees on either:

  1. the message directory, or

  2. the message state, or

  3. both.

4.14.7 DIAG

The program DIAG runs in a stand-alone configuration. It makes use of the SIGMA terminal driver as loaded by SIGMA without modification. It is designed to allow test driving a SIGMA terminal by generating dispatches by hand or by sequencing through an existing terminal trace file. For consistency, the DIAG program must live in the same directory as the Driver SAV file, namely in <MME-RUNTIME> (the Driver SAV file is taken at runtime from the directory in which DIAG itself lives). Since at CINCPAC the contents of <MME-RUNTIME> are dumped onto another directory corresponding to the SIGMA release (e.g., <SIGMA-2200>), a convenient way to avoid having to remember (and type) the current residence directory of DIAG is to place a calling program on <SUBSYS> which is changed at each new SIGMA release to point to the latest release of DIAG.

When DIAG starts, it first determines whether there is a TTY already assigned to the job. If so, the user is asked to confirm using that TTY. If he confirms negatively or there is no previous TTY assignment, he is prompted for the TTY line number of the HP/MME terminal to be diagnosed. Next, DIAG prompts for the file name into which the Terminal Driver's output is to be put (especially important for the DUMP command, described below); if defaulted by the user's entering <Escape> or <Carriage-Return>, DIAG will name this file DIAG.TRACE on the connected directory.

The program is controlled by single-keystroke commands which are prompted by one-line menus of command words. The interface should be obvious. Since the controlling terminal is the one used to control DIAG and is also used for notice feedback by the driver, notices may appear interspersed among commands. This should not prove overly distracting.

NOTE: All numeric input to DIAG is in octal, for consistency with trace files. Editing of numeric input is not provided, but each number entered must be confirmed by a <Space> or <Carriage-Return>. If an incorrect digit is entered, simply hit a <DEL> or any other non-confirming character and DIAG will echo "XXX" and wait for another number.

Command codes are those recognized by the terminal (rather than those used in the routines in TISUB), but they must be typed in octal whereas the terminal specification gives them in decimal.

There is a switch in the Driver (module DV2FRK) which checks the common cell ?STATE.RELEASE before sending a Reset dispatch to the terminal. If SIGMA (or XSIGMA) is running, the Release number will be nonzero and a Reset dispatch is sent. If the DIAG program has started up the Driver, the Release number will NOT have been set (it will be zero) and the Reset is inhibited. A Protocol Reset is done in either case on startup.

DIAG allows dispatches to be generated in several ways:

1. VERBATIM -
A dispatch is typed as a sequence of octal bytes. The first is the total dispatch length (counting the length byte itself); the second is the command code; subsequent bytes are arguments of the given command, as required. Any sequence of bytes can be generated this way and passed verbatim to the terminal. This allows generating illegal dispatches.

2. TISUB -
The first byte is the total dispatch length as in the VERBATIM case, and the second byte is the command code. However, the command code is interpreted and a sequence of prompts is generated for the appropriate arguments to the indicated TISUB routine. Only legal dispatches can be passed this way, since TISUB does its own error checking. This method of generating dispatches simulates the way SIGMA itself generates them. *In this mode, if an illegal* command code (e.g., zero) is specified, a menu of allowed commands is shown, with associated octal codes. Dispatches requiring text arguments allow ASCII text to be typed, with the length of the text string implied by the initial total dispatch-length byte.

3. From File -
Files of the format generated by SIGMA to trace the terminal dialog (including both Dispatches and Notices) may be used to drive DIAG. The files SIGMA generates for such traces are named {USER-STATE/SIGMA}.TRACE, but any file of the right format may be used. These files are assumed to consist of the words DISPATCH or NOTICE at the beginning of the first line of a Dispatch or Notice, respectively. DIAG steps through such a file ignoring Notices.

In File mode, there are several Options provided, including simple commands to turn on or off Dispatch Tracing and to Quit the file. Dispatch Tracing (initially ON) echoes each dispatch sent from the Input file on the controlling terminal, to make it more obvious what is going on. It can be turned off to allow sending a large group of dispatches without feedback. Quitting from a file takes DIAG back to its top level and closes and releases the open Dispatch Input file.

The primary options for File Input are:

- Continue - Typing "C" or a Carriage Return steps through the file and sends the next Dispatch.

- Sequence - Typing "S" prompts for a range of SEQuence numbers greater than or equal to zero (the first Dispatch in a trace file is SEQ: 0). DIAG steps through Dispatches looking for the "SEQ:" trace and compares the sequence number that follows to see if it is within the specified range. When all Dispatches in the range have been sent, DIAG remains in the File mode and prompts for file Options again.

At the top level, DIAG provides the following modes:

Continue(CR)	"C" or Carriage Return continues in the same mode.
Tisub	"T" generates subsequent dispatches in TISUB mode.
Verbatim	"V" generates subsequent dispatches in VERBATIM mode.
File	"F" switches to File mode and prompts for File Options.
Repeat	"R" resends the previous dispatch, however generated.
Dump	"D" allows dumping the contents of a hung terminal.

The Dump option allows dumping the contents of a hung or crashed HP/MME terminal. The hung terminal line must be assigned to another controlling terminal which then runs DIAG and asks for the Dump option. Note that a Driver Protocol Reset is done when DIAG starts up the Driver, but that the terminal's memory itself is NOT Reset (the Driver omits this Terminal Reset when it is not being run by SIGMA itself).

The user is warned that dumping will hang both the controlling and the assigned terminals for something on the order of 25 minutes. If he confirms that he wants the dump, a special sequence is sent to the terminal which initiates a dump using the Protocol. This generates Notices which go into the Dispatch/Notice trace file, each containing an encoding of a part of the terminal's memory.

REFERENCES

1. Abbott, R. J., *A Command Language Processor for Flexible Interface Design*, USC/Information Sciences Institute, ISI/RR-74-24, 1974.

2. Ames, S. R., and W. W. Plummer, *TENEX Security Enhancements*, MITRE Corporation, Technical Report MTR-3217, April 1976.

3. Ames, S. R., and D. R. Oestreicher, "Design of a message processing system for a multilevel secure environment," in *Proceedings of the National Computer Conference*, AFIPS,, 1978. Also appeared as Mitre Corporation Technical Report MTR-3449, June 1978.

4. Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a paged time sharing system for the PDP-10," *Communications of the ACM* 15, (3), March 1972, 135-143.

5. AUTODIN message CINCPAC 070200Z. August 1975.

6. Military Message Experiment Selection Criteria, 17 September 1976. Prepared by CTEC, Inc., 7777 Leesburg Pike, Falls Church, Virginia 22043.

7. DISTAN Program. Prepared by Naval Electronic Systems Command, Material Acquisition Directorate Telecommunication Division, July 1976.

8. Ellis, T. O., L. Gallenson, J. F. Heafner, and J. T. Melvin, *A Plan for Consolidation and Automation of Military Telecommunications on Oahu*, USC/Information Sciences Institute, ISI/RR-73-12, May 1973.

9. Goodwin, N. C., J. Mitchell, and P. S. Tasker, *Evaluation of ARPANET Message-Handling Systems for Use by the Military*, MITRE Corporation, Technical Report MTR-3096, August 1975.

10. Goodwin, N. C., J. Mitchell, and P. S. Tasker, *Concept of Operations for Message-Handling in CINCPAC*, MITRE Corporation, Technical Report MTR-3323, October 1976.

11. Goodwin, N. C., J. Mitchell, and S. W. Slesinger, *Test Plan for Military Message Handling Experiment*, MITRE Corporation, Technical Report MTR-3268, July 1976.

12. Goodwin, N. C., *Military Message Experiment Baseline Data Report Test Group*, MITRE Corporation, Technical Report MTR-3665, September 1978.

13. Goodwin, N. C., and S. W. Hosmer, *A User-Oriented Evaluation of Computer-Aided Message Handling*, MITRE Co., MTR 3920, April 1980. (MME Final Report, Volume VI, Part 1. [29])

14. Goodwin, N. C., and S. W. Hosmer, *Appendices to a User-Oriented Evaluation of Computer-Aided Message Handling*, MITRE Co., MTR 3946, April 1980. (MME Final Report, Volume VI, Part 2. [29])

15. Heafner, J. F., *A Methodology for Selecting and Refining Man-Computer Languages to Improve Users' Performance*, USC/Information Sciences Institute, ISI/RR-74-21, 1974.

16. Heafner, J. F., *Analysis of Man-Computer Languages: Design and Preliminary Findings*, USC/Information Sciences Institute, ISI/RR-75-34, 1975.

17. Heafner, J. F., and L. H. Miller, *Design Considerations for a Computerized Message Service Based on Tri-Service Operations Personnel at CINCPAC Headquarters, Camp Smith, Oahu*, USC/Information Sciences Institute, Technical Report ISI/WP-3, September 1976.

18. Holg, Chloe, *The Military Message Experiment SIGMA Primer*, USC/Information Sciences Institute, 1977. ISI/TM-77-9.

19. -----, *2645A Display Station Reference Manual*, Hewlett-Packard Company, , 1976.

20. IA Project, Military Message Processing System Design. Unpublished design document. 10 January 1975.

21. Intel, *Intel 8080 Microcomputer System's User's Manual*, Intel Corporation, Technical Report, 1975.

22. Kallander, J. W., N. C. Goodwin, S. Hosmer, C. Smith, D. Fralick, L. Klitzkie, and S. H. Wilson, *Military Message Experiment Mid Experiment Report*, Naval Research Laboratory, NRL Memorandum Report 4094, November 1979.

23. Mandell, R. L., *An Executive Design to Support Military Message Processing Under TENEX*, USC/Information Sciences Institute, ISI/RR-74-25, 1975. draft only

24. Miller, D., *Military Message Handling Experiment Training Requirements*, MITRE Corporation, Technical Report MTR-3263, June 1976.

25. Miller, David G., *Military Message Experiment Training Experience*, MITRE Corporation, Technical Report MTR-3644, August 1978.

26. Miller, D. G., *MME - Final Training Report*, MITRE Corporation, Bedford, Mass., Technical Report MTR-3919, May 1980.

27. -----, Memorandum of Agreement between Director, Defense Advanced Research Projects Agency (DARPA), Commander, Naval Telecommunications Command (NAVTELCOM), Commander, Naval Electronic Systems Command (NAVELEX), and Commander-in-Chief, Pacific (CINCPAC), 1975. Unpublished memorandum.

28. House Appropriations Committee, Report 95-451. U.S. Congress. 21 June 1977

29. MME Final Report. The *MME Final Report* is being prepared by various individuals and organizations involved in the MME. It will consist of eight volumes; some of the volumes themselves consist of more than one part. References [13] and [14] are Volume VI. For information about how to obtain the other volumes of the *MME Final Report*, contact the Naval Research Laboratory, Washington, D.C. 20375, Attn: Code 7503.

30. *Naval Telecommunications Procedures, Telecommunications Users Manual, NTP3*, 4401 Massachusetts Ave., N.W., Washington, D.C. 20390, 1974.

31. Oestreicher, D., P. Raveling, and R. Stotz, *HP/MME Terminal - Application Specification*, USC/Information Sciences Institute, Technical Report ISI/TM-78-10, March 1978.

32. Rothenberg, J. G., *An Intelligent Tutor: On-line Documentation and Help for a Military Message Service*, USC/Information Sciences Institute, Technical Report ISI/RR-74-26, May 1975.

33. Rothenberg, J. G., *An Editor to Support Military Message Processing Personnel*, USC/Information Sciences Institute, ISI/RR-74-27, June 1975.

34. Rothenberg, J., *DARPA Navy CINCPAC Military Message Experiment SIGMA Message Service Reference Manual*, USC/Information Sciences Institute, Technical Manual 78-11.2, June 1979.

35. Rothenberg, J., "On-line tutorials and documentation for the SIGMA Message Service," in *Proceedings of the National Computer Conference*, AFIPS, June 1979.

36. Slesinger, S. W., and N. C. Goodwin, *Test Procedures for Military Message-Handling Experiment*, MITRE Corporation, Technical Report MTR-3521, October 1977.

37. Oestriecher, D., et al., SIGMA Transition and Deficiency Amelioration Plan, 1977. Unpublished note.

38. Stotz, R., R. Tugender, D. Wilczynski, and D. Oestreicher, "SIGMA -- An interactive message service for the Military Message Experiment," in *Proceedings of the National Computer Conference*, AFIPS,, June 1979.

39. Stotz, R., P. Raveling, and J. Rothenberg, "The terminal for the Military Message Experiment," in *Proceedings of the National Computer Conference*, AFIPS, June 1979.

40. Tangney, J. D., S. R. Ames, and E. L. Burke, *Security Evaluation Criteria for MME Message Service Selection*, MITRE Corporation, Technical Report MTR-3433, June 1977.

41. Tangney, John D., *MME Security Test Procedures*, MITRE Corporation, Technical Report MTR-3615, June 1978.

42. Tugender, R., and D. R. Oestreicher, *Basic Functional Capabilities for a Military Message Processing Service*, USC/Information Sciences Institute, Technical Report ISI/RR-74-23, 1975.

43. Tugender, R., "Maintaining order and consistency in multi-access data," in *Proceedings of the National Computer Conference*, AFIPS,, June 1979

44. Wilczynski, D., R. Tugender, and D. Oestreicher, "The SIGMA experience -- A study in the evolutionary design of a large software system," in *Proceedings of the National Computer Conference*, AFIPS,, June 1979.

45. Wilczynski, D., R. Stotz, R. Tugender, and R. Lingard, "Message system architecture -- Experience at CINCPAC with the SIGMA System," in *Compcon Spring '80*, IEEE, February 1980.

46. Wilson, S. H., J. W. Kallander, N. M. Thomas III, L. C. Klitzkie, and J. R. Bunch, Jr., *Military Message Experiment Quick Look Report*, Naval Research Laboratory, NRL Memorandum Report 3992, April 1979.

47. Wulf, W. A., D. B. Russell, and A. N. Habermann, "BLISS: A language for systems programming," *Communications of the ACM* 14, (12), December 1971, 780-790.

INDEX

PRECEDING PAGE BLANK-NOT FILMED

FILMED

8-8